Proceedings of CICLOPS'2003

Colloquium on Implementation of Constraint and LOgic Programming Systems

December 2003

Ricardo Lopes, Michel Ferreira (Eds.)

Technical Report DCC-2003-05

Departamento de Ciência de Computadores
Faculdade de Ciências
Universidade do Porto

PREFACE

This book contains the Proceedings of the CICLOPS'03 – Colloquium on Implementation of Constraint and LOgic Programming Systems 2003 – held in Mumbai (India), December 2003.

CICLOPS'03 means to bring together, in an informal setting, people involved in research on sequential and parallel implementation technologies for logic and constraint programming languages and systems, in order to promote a much needed exchange of ideas and feedback on recent developments. We hope that the workshop will provide meeting ground for people working on implementation technology for different aspects of execution of logic-based and constraint-based languages and systems.

This workshop continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with considerable success in Budapest (1993) and Ithaca (1994), the Compulog Net workshops on Parallelism and Implementation Technologies held in Madrid (1993 and 1994), Utrecht (1995) and Bonn (1996), the Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages held in Port Jefferson (1997), and Manchester (1998), Las Cruces (1999), and London (2000), and recently the Colloquium on Implementation of Constraint and LOgic Programming Systems in Paphos (Cyprus-2001) and Copenhagen (2002).

Workshop Coordinators

Michel Ferreira and Ricardo Lopes (Portugal)

Program Committee

- Bart Demoen (Belgium)
- Christian Schulte (Sweden)
- David S. Warren (USA)
- Enrico Pontelli (USA)
- Haifeng Guo (USA)
- Inês de Castro Dutra (Brazil)
- Kostis Sagonas (Sweden)
- Manuel Carro (Spain)
- Michel Ferreira (Portugal)
- Ricardo Lopes (Portugal)

Referees

Bart Demoen, Bert Van Nuffelen, Christian Schulte, David S. Warren, David Trallero Mena, Enrico Pontelli, Hai-Feng Guo, Inês de Castro Dutra, José F. Morales, Kostis Sagonas, Luís Lopes, Manuel Carro, Manuel Eduardo Correia, Michel Ferreira, Nikolay Pelov, Tom Schrijvers, Ricardo Lopes, Ricardo Rocha

Acknowledgments

This printing was supported by project APRIL (Project POSI/SRI/40749/2001).

CICLOPS 2003 Program

Performance Issues in Prolog Applications.

Vítor Santos Costa
COPPE/Universidade Federal do Rio de Janeiro, Brasil
vitor@cos.ufrj.br
http://www.cos.ufrj.br/∼vitor

Abstract

Prolog is an expressive programming language based on a subset of First Order Logic, and has been widely used in Artificial Intelligence Research. Examples include Machine Learning, say, for implementing Inductive Logic Programming, and Natural Language Processing, where applications range from the well-known work in Definite Clause Grammars to automata-based parsing. In this talk, we discuss how Prolog implementations matter in achieving AI application performance and scalability, and present some solutions that are currently being research for Prolog systems. Throughout we draw from our own experience in supporting a Prolog system, and in designing ILP applications.

We observe that excellent data-base indexing is critical. Often, applications are initially developed for smallish examples, where data-base access is guaranteed to be fast. Unfortunately, when practitioners experiment with real-life like situations, performance simply breaks down. Until recently, Prolog implementations were simply not good enough in this regard. We discuss some recent work on the B-Prolog, XSB and YAP systems that tries to address the problem.

A related issue is that efficient data-base updating must be supported. For instance, many AI systems must perform search, and use the data-base to store the search space. Many AI applications thus add and remove items often, whilst still requiring fast lookup. We discuss two solutions: the use of tries as in XSB Prolog, and extending the indexing mechanism, as done recently in YAP Prolog.

Some Prolog applications require functionality beyond what is provided by the standard Prolog engine. Sometimes, we will need Prolog extensions, but sometimes even better performance can be achieved by writing a specialised interpreter or performing a simple transformation. In one example, we discuss a small program for modelling RAS pathways on a cell. Tabling allows a declarative formulation to run for non-trivial queries. But full tabling is not necessary: a poor-man's version of tabling achieves even better results. As a second example we discuss the CLP($\mathcal{BN}$) implementation: although we need coroutining to support generic queries, we have taken advantage of meta-interpreters, for instance, to do efficient learning when complete data is available.

We believe that Prolog can be effectively used for non-trivial AI programs. To do so, collaboration between users and implementors is fundamental.

CHR for XSB

Tom Schrijvers David S. Warren Bart Demoen
Dept. of Computer Science, K.U.Leuven, Belgium
Dept. of Computer Science, State University of New York at Stony Brook, USA
{toms,bmd}@cs.kuleuven.ac.be warren@xsb.com

Abstract

XSB is a highly declarative programming system consisting of Prolog extended with tabled resolution. It is useful for many tasks, some of which require constraint solving. Thus flexible and high level support for constraint systems is required. Constraint Handling Rules is exactly such a high level language embedded in Prolog for writing application tailored constraint solvers.

In this paper we present the integration of a CHR system in the XSB system and especially our findings on how to integrate CHR with tabled resolution, such as how to deal with issues as call abstraction of constraints, constraint store merging, answer store projection and constraint store representations for tabling.

We illustrate the power of the XSB-CHR combination with two examples in the field of model checking. It is indeed possible to quickly write application specific constraint solvers, experiment with them and achieve a reasonable performance and high readability. The combination of XSB's goal-driven fixpoint execution model with CHR's committed choice bottom-up approach has proven not only feasible, but considerably useful as well.

1 Introduction

XSB (see [14]) is a Prolog system with tabled resolution. Tabled resolution is useful for recursive query computation, allowing programs to terminate in many cases where Prolog does not. Parsing, program analysis, model checking, data mining and diagnosis and many more applications benefit from tabled resolution. We refer the reader to [1] for a coverage of XSB's SLG execution strategy.

The use of constraint solvers in XSB has been a quite laborious and inconvenient endeavor up to now. Initially XSB provided no builtin support at all for dealing with constraints. Hence XSB programmers resorted to interfacing with foreign language libraries or implementation of constraint solvers in XSB itself with close coupling of constraint solver and application as a consequence. The initial feasibility study of a real time model checking system used a meta interpreter written in XSB to deal with constraints (see [11]). The full system implementation then reports of interfacing XSB with the POLINE polyhedra based constraint solver library and passing around handles to the constraint store in the XSB program (see [7]). A later version of this real time model checking application switched to using distance bound matrices implemented in XSB itself (see [12]). This shows that there is certainly a demand for constraints in the XSB setting, but that a satisfactory solution with sufficient ease of use and a reasonable implementation has not been found so far.

In an attempt to amend some of the constraint problems in XSB, it has been extended with attributed variables (see [2]). Attributed variables is a Prolog language feature that is particularly suited for constraint solver implementation as it allows efficient association of data with variables and user hooks on variable binding. Unfortunately this feature has not caught on in XSB as a basis for constraint systems because it is a particularly low level feature that still requires considerable scheduling considerations by the constraint solver programmer. However, the work on attributed variables in XSB is not lost, as attributed variables are indeed a powerful implementation tool for constraint systems: efficient compilation of CHR to Prolog relies heavily on it (see [10]).

It is precisely these key features of CHR that are missing in XSB: Constraint Handling Rules, or CHR for short, is a high level language designed for writing application-oriented constraint systems (see [8]). We refer the reader to [9] for a survey of syntax, semantics, theoretical and practical work on CHR. In this paper we will introduce CHR with a quick informal review of syntax and semantics using a simple example (see Section 1.1).

Section 2 will present a general overview of the hProlog CHR system as well as some of its more interesting implementation details. Section 3 will then address the core matter of this paper, the integration of the CHR system in tabled execution. Subsequently, Section 4 briefly illustrates the power of the resulting CHR-XSB system with two model checking applications, one of which is the earlier mentioned real time model checking application. Finally, Section 5 concludes and suggests possible future work.

1.1 CHR by Example

The set of constraint handling rules below defines a less-than-or-equal constraint (leq/2) over numbers. The rules illustrate several syntactical features of CHR.

$$\begin{array}{l}
\text{X leq X} \Longleftrightarrow \text{true.} \\
\text{X leq Y} \Longleftrightarrow \text{number(X), number(Y)} \mid \text{X =< Y.} \\
\text{X leq Y, Y leq X} \Longleftrightarrow \text{X = Y.} \\
\text{X leq Y} \setminus \text{X leq Y} \Longleftrightarrow \text{true.} \\
\text{X leq Y, Y leq Z} \Longrightarrow \text{X leq Z.}
\end{array}$$

The first, second and third rule are simplification rules, indicated by the double arrow. To the left of the arrow is the head of a rule. A simplification rule has the meaning that the constraints in the head can be simplified to the Prolog goal in the body, `true` for the first rule. Variables in constraints are never bound to each other or to terms in the head of a rule; only equality tests are used. The meaning of this first rule should be obvious: the leq relation is reflexive, and hence `X leq X` is trivially satisfied and bears no information.

The second rule shows that a rule can be extended by a guard, after the arrow and before the vertical bar. In this case the guard is `number(X), number(Y)`. The body of the rule is only executed for constraints that match the head and satisfy the guard. The guard can be any Prolog goal that does not bind variables of the head. Rule two replaces the constraint with a simple Prolog inequality check if the arguments are bound to numbers.

The third rule illustrates that the head of a rule can contain a conjunction of multiple constraints. It formulates the antisymmetry property of the leq constraint.

The fifth rule with the $\Longrightarrow$ is a propagation rule. The body of the rule is executed once for every matching combination of constraints in the head, not removing the head constraints.

The fourth rule is a “simpagation” rule. It has the same meaning as a simplification rule where the constraints before the backslash would be posed again in the body. However it is more efficient in that it never removes those head constraints and does not unnecessarily trigger rules in that way. In the leq constraint definition its role is to declare the set semantics of the constraint, i.e. the number of copies of a constraint is not important and hence it is more efficient to keep only one.

Operationally, when a constraint is posed the rules are tried in order. For a multi-headed rule, the additional constraints are looked for in the constraint store. The outcome is that the posed constraint is either simplified away or reaches the end of the rules and gets suspended in the constraint store. Suspended it can either be used as an additional constraint for a multi-headed rule or wait until it gets triggered. Triggering of a suspended constraint occurs when any variable in the constraint gets bound. The constraint then tries to match all rules in order again.

2 The hProlog CHR System

Initially the CHR system described in this paper was written for the hProlog system. hProlog is based on dProlog (see [6]) and intended as an alternative backend to HAL (see [5]) next to the current Mercury backend. The initial intent of the implementation of a CHR system in hProlog was to validate the underlying implementation of dynamic attributes (see [4]).

The hProlog CHR system consists of a preprocessor and a runtime system. The preprocessor compiles embedded CHR rules in Prolog program files into Prolog code.

The compiled form of CHR rules is very close to that of the CHR system by Christian Holzbaur, which is used in SICStus and Yap. The precompiler is intended as a basis for experimentation with optimized compilation of CHR rules, both through inference and programmer declarations.

The runtime system is nearly identical to that of Christian Holzbaur: suspended constraints are stored in a global constraint store. Variables in suspended constraints have attributes on them that function as indexes into this global store. Binding of these attributed variables causes the suspended constraints on them to trigger again.

The main advantage of the hProlog implementation is that the dynamic nature of the attributed variables in hProlog allows to move more functionality from the compiled rules to the runtime system.

Little difficulty was experienced while porting the preprocessor and runtime system from hProlog to XSB. The main problem turned out to be XSB's overly primitive implementation of attributed variables: it did not support attributes in different modules. Moreover, the actual binding of attributed variables was being delayed to the interrupt handler where it was left up to the programmer. This causes unintuitive and unwanted behavior in several cases: while the binding is delayed from unification to interrupt handling, other code can be executed in between that relies on variables being bound, e.g. arithmetic. Due to these problems of the current XSB attributed variables, it was deemed acceptable to model them more closely to the hProlog behavior. This of course facilitated the porting of the CHR system considerably.

3 CHR and Tabled Execution

The main challenge of introducing CHR in XSB is integrating the forward chaining fixpoint computation of the CHR system with the backward chaining fixpoint computation of tabled resolution.

A similar integration problem has been solved in [2], where a general framework for constraint solvers written with attributed variables for XSB is described. The name Tabled Constraint Logic Programming (TCLP) is coined in that publication.

The main difference for the programmer between CHR and attributed variables for developing constraint solvers, i.e. the fact that CHR is a much higher level language, should be carried over to the tabled context. Hence tabled Constraint Handling Rules should provide a more convenient level of programming constraint solvers, hiding execution details whenever possible, than TCLP with attributed variables.

In [2] the general framework specifies three operations to control the tabling of constraints: call abstraction, entailment checking of answers and answer projection. It is left up to the constraint solver programmer how to implement these operations with respect to his solver implementation.

In the following we formulate these operations in terms of the CHR implementation and provide a higher level CHR interface for answer projection. In this manner the solver programmer is not confronted with the underlying CHR implementational intricacies.

3.1 Call Abstraction

Call abstraction replaces the called goal with a call to a more general goal followed by an operation that ensures that only the answer substitutions applicable to the particular call are retained. At the level of ordinary non-constraint Prolog, abstraction means not passing certain bindings in to the call. E.g. `p(q,A)` can be abstracted to `p(Q,A)`. This goal has then to be followed by `Q = q` to ensure that only the appropriate bindings for `A` are retained.

In XSB call abstraction is a means to control the number of tables. When a predicate is called with many different instantiation patterns, a table is generated for each such call instantiation pattern. Thus it is possible that the information for the same fully instantiated call is present many times in tables for different call instantiation patterns. However, this amount of duplication in the tables can be avoided by using call abstraction to restrict to a small set of call instantiation patterns.

For constraint logic programming, call abstraction can be extended from bindings to constraints: abstraction means removing some of the constraints on the arguments. Consider for example the call `p(Q,A)` with constraint `Q leq N` on `Q`. This call can be abstracted to `p(Q',A)`, followed by `Q' = Q` to reintroduce the constraint.

Abstraction is especially of value for those constraint solvers where the number of constraints on a variable can be much larger than the number of different bindings for that variable. Consider for example a finite domain constraint solver with constraint `domain/2`, where the first argument is a variable and the second argument the list of its possible values. If the variable can be bound to at most n values it can take as much as 2^n different `domain/2` constraints, one for each subset of values.

Varying degrees of abstraction are possible and may depend on the particular constraint system or application. Full constraint abstraction, i.e. the removal of all constraints from the call, is generally more suitable for CHR for the following reason:

- CHR rules do not require constraints to be on variables. This means that constraints can be on ground terms or atoms as well. It is not straightforward to define abstraction for ground terms as these are not necessarily passed in as arguments but can just as well be created inside the call. Hence there is no explicit link with the call environment, while such a link is needed for call abstraction. As such, only no abstraction or full constraint abstraction seem suitable for CHR.

- Full constraint abstraction is preferable when the previously mentioned table blow-up is likely.

Moreover, it may be quite costly for certain constraint domains to sort out what constraints should be passed in to the call or abstracted away, involving transitive closure computations of reachability through constraints. Hence often full abstraction is cheaper.

For CHR full abstraction requires the execution of the tabled predicate with a fresh empty constraint store. If the call environment constraint store were used, interaction with new constraints would violate the assumption of full abstraction.

The code below shows how a predicate `p/1` that requires tabling:

```
:- table p/1.

p(X) :- ...
```

is transformed into two predicates, where the first one is called, takes care of the abstraction, calls the second predicate and afterwards combines the answer with the previously abstracted away constraints.

```
p(X) :-
        get_global_store(S_E),
        set_empty_store,
        tabled_p(X1,S_A),
        merge_stores(S_E,S_A,S_E1),
        set_global_store(S_E1),
        X1 = X.

:- table tabled_p/2.

tabled_p(X,S_A) :- ...
```

The further implementation of `tabled_p` and `merge_stores` will be discussed in the next Sections.

3.2 Tabled Store Representation and Merging

When a tabled predicate `p` returns, the answer constraint store s_a should be stored in the answer table. When a subsequent call to `p` is made, s_a should be fetched from the table and merged with the calling environment constraint store s_e. On a high level this means that all the constraints in s_a have to be inserted in s_e and triggered in such a fashion that the merged store reaches a consistent final state, i.e. with all the applicable simplifications and propagations.

The implementation of the tabled predicate `tabled_p` mentioned above is revealed here. It has an extra argument, the tabled store representation `S_A`, that is extracted from the global store after the original body of `p`, now moved to the predicate `orig_p`, has been executed.

```
tabled_p(X,S_A) :-
        orig_p(X),
        extract_store_representation(S_A).

orig_p(X) :- ... /* body of original p */.
```

The representation of the answer store in the table and the highly correlated merging algorithm determine largely the cost of a call to a tabled predicate with constraints. Two different implementations have been explored and it appears the more naive approach is the best. An indication of what programs or predicates are good candidates for tabling with respect to time efficiency, is obtained from this conclusion. Obviously if termination is an issue, tabling is paramount.

Suspension Representation This representation aims at storing the suspended constraints in the answer table in much the same way as they are represented in the constraint store. The constraint store is an updatable term, containing suspended constraints grouped by functor. Each suspended constraint is represented as a suspension term, containing among others the following information:

- the unique ID for sorting and equality testing
- the goal to execute when triggered, this goal contains the suspension itself as an argument, hence creating a cyclic term
- the propagation history

Furthermore, variables involved in the suspended constraints behave as indexes into the global store: they have the suspensions stored in them as attributes.

It is possible to store attributed variables in answer tables (see [3]), but two other issues do pose a problem. Firstly, the tables do not deal with cyclic terms[1]. This can be dealt with by breaking the cycles before storage and resetting them after fetching. Secondly, the unique identifiers have to be replaced after fetching by fresh ones as multiple calls would otherwise create multiple copies of the same constraints all with identical identifiers.

In addition to the above operations upon retrieval, the suspensions are inserted into the call environment store and then triggered.

By keeping the tabled representation as close as possible to the global store representation we hope to save on execution time during merging: the propagation history of the answer store is retained and hence answer constraints will not propagate a second time.

It turns out that in the programs we have tested so far, preparing constraints for storage in the tables and proper initialization upon retrieval is considerably more costly than repropagation. One has to bear in mind that retrieval of a table consists of creating new terms and attributed variables. Hence preserving a useful structure in the table is hardly better than preserving the data in any other format that allows easy derivation of that structure.

Naive Representation The naive representation aims at keeping the information in the table in as simple a form as possible: for each constraint only the goal to pose this constraint is retained in the table. It is easy to create this goal from a suspension and easy to merge this goal back into another constraint store: it needs only to be called.

When necessary the goal will create a suspension with a fresh unique ID and insert it into the constraint store. However in many cases it may prove unnecessary to do so because of some simplification through interaction with constraints in the calling environment.

The only information that is lost in this representation is the propagation history. This may lead to multiple propagations for the same combination of head constraints. For this to be sound, it is necessary that the CHR rules behave according to set semantics, i.e. the presence of multiple identical constraints should not lead to different answers modulo identical constraints.

In all the applications we have encountered, this approach turns out to be better. The simplicity of storage and retrieval are more important than unnecessary propagation overhead.

[1]If the cycle point were represented as an attributed variable, then XSB tabling would handle the cyclic terms. However, this representation was deemed inappropriate due to its complexity and expected performance.

All in all the conclusion seems to be that superior efficiency through tabling can only be achieved if a certain amount of simplification or non-constraint related computation occurs inside the tabled predicate and if the cost of creating the tabled constraints is smaller than executing the predicate. Hence just as it makes no sense to table `append/3`, it makes no sense to table the constraint equivalent of `append/3`, a predicate that builds a large constraint store straightforwardly.

3.3 Answer Combination and Entailment Checking

In some cases it is undesirable to have multiple answers for a tabled predicate. While all the answers are valid, they may all be just approximations. In such a case one would like to combine all answers to a single most specific answer.

Using the XSB local strategy for table completion, at the end of the tabled predicate we merge a previous answer store s_0 with a new answer store s_1. After merging the store will be simplified and propagated to s, combining both answers. If this combined answer s is different from s_0, then s_0 is discarded.

The computation of the shortest path serves as a good illustration:

path(A,B,D) :- edge(A,B,D1), D leq D1.
path(A,B,D) :- path(A,C,D1), edge(C,B,D2), D leq D1 + D2.

Suppose appropriate rules for the leq/2 constraint in the above program, as in Section 1.1. The query path(x,y,D) will then find an answer for every single path from `x` to `y`. The answers will only differ in the upper bound on `D`.

If we are only interested in the most specific answer, we can make sure to include the following CHR rule:

$$\text{X leq D1} \setminus \text{X leq D2} \Longleftrightarrow \text{D1} =< \text{D2} \mid \text{true.}$$

The same mechanism can be used to check entailment: if the combined answer store s is equal to one of the two, then that answer entails the other.

$$s_0 + s_1 = s_i \wedge i \in \{0, 1\} \Longrightarrow s_i \vdash s_{1-i}$$

Here the symbol + is used to indicate merging of constraint stores and = means equality of constraint stores. Constraint store equality is discussed later, in Section 3.5

3.4 Answer Projection

Often it is necessary to project the answer constraint store on the non-local variables of the call. The usual motivation is that constraints on local variables are meaningless outside of the call. The constraint system should be complete so that no unsatisfiable constraints can be lost through projection.

For tabling there is an additional and perhaps even more pressing motivation for projection: a predicate with an infinite number of different answers may be turned into a predicate with a finite number of constraints by throwing away the constraints on local and unreachable variables.

In some cases it may suffice to look at the constraints in the store separately and given a set of non-local variables to decide whether to keep the constraint or not. In those cases it may be convenient to exploit the operational semantics of CHR rules and implement projection as a `project/1` constraint with the list of variables to project on as an argument. A series of simpagation rules can then be used to look at and decide on what constraints to remove. A final simplification rule at the end can be used to remove the `project/1` constraint from the store.

The predicate `tabled_p` would then look like:

```
tabled_p(X,S_A) :-
        orig_p(X),
        project([X]),
        extract_store_representation(S_A).
```

The following example shows how to project away all leq/2 constraints that involve arguments not contained in a given set $VarSet$:

project(VarSet) \ X leq Y $\Longleftrightarrow$ \+ (member(X,VarSet),member(Y,VarSet)) | true.
project(VarSet) $\Longleftrightarrow$ true.

Besides removal of constraints more sophisticated operations such as weakening are possible. E.g. consider an constraint $\in/2$ for a set constraint solver that constraints an element to be in a list and a *nonempty*/1 constraint that indicates a set should not be empty:

project(VarSet) \ Elem $\in$ Set $\Longleftrightarrow$ member(Set,VarSet), \+ member(Elem,VarSet) | nonempty(Set).

This approach is of course not general in the sense that certain constraint domains may need more information than just the variables to project on, such as more intricate knowledge of the contents of the constraint store. In addition it relies on operational semantics and ordering of constraints. However, it is a rather compact and high level notation and as such it might be possible to infer conditions on its usage under which the technique is provably correct.

3.5 Constraint Store Equality

The need to check constraint store equality arises at three different locations:

- Partial call abstraction means a subset of the call environment store is passed in. The tabling system then needs to check whether a previous call with the same passed in constraint store appears in a table.
- Entailment checking means we need to check whether a merged store equals one of the initial stores.
- New answer checking means that a new answer store is compared with previous answer stores. This operation performed by the tabling mechanism is needed to avoid multiple copies of the same answer.

We can consider this equality checking with the naive representation of constraints presented previously in mind. Any permutation of this list represents the same constraint store.

When exact variable identity matters, i.e. in the case of entailment checking, the representation can be easily brought in a canonical form based on an arbitrarily chosen ordering of the involved variables, e.g. by sorting of the constraints.

For comparison with call or answer patterns in the tables, exact variable identity is not required. Equality checking needs to be done modulo variable renaming.

Elaboration on heuristics for an algorithm falls outside of the scope of this paper. The problem can be ignored altogether, with possible duplication in tables as a consequence, or only partially tackled, e.g. by simple sorting and pattern matching.

4 Applications

XSB Prolog, with its tabling, has proven very convenient for model checker implementation. The XMC toolset (see [13]), written on top of XSB, is a witness to that. An important feature previously missing from XSB, a simple way to write application tailored constraint solvers, has now been added with CHR.

In this section we will look at two model checking applications that both use tabling and constraint solving but in different ways. Both systems previously turned to ad hoc implementations of constraint solvers to satisfy their constraint solving needs. It took very little effort to replace these ad hoc solvers with more flexible and higher level CHR solvers.

We will focus on the constraint-related problems only and refer the reader to specialized publications regarding model checking, if more insight into the bigger scope of the applications is desired. However the common approach of the two following model checking applications is based on reachability between states in an automaton or nodes in a graph.

The standard reachability definition in Prolog gives rise to infinite loops for cyclic graphs.

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- edge(X,Z), reach(Z,Y).
```

Fortunately in XSB these infinite loops are avoided by tabling the `reach/2` predicate. Hence this `reach/2` predicate is the main intersection point for tabling and CHR in our two applications.

4.1 Model Checking of Data-Independent Systems

Data-independent systems manipulate data variables over unbounded domains but have a finite number of control locations. Such systems can be modeled as extended finite automata, finite automata with guards on the transitions and variable mapping relations between source and destination locations.

The approach of [15] represents these systems as constraint logic programs: variables are passed along states collecting more and more constraints. Hence the `reach/2` predicate will have to be described as discussed in Section 3 to deal with CHR constraints on the relevant variables.

Of this constraint approach to checking data-independent systems we studied one model that checks for a particular vulnerability in the comsat program. Comsat is a Unix server that notifies users of new mail by printing the first 7 lines to the user's terminal. Earlier versions of this program had a vulnerability that would allow a malicious person to obtain root access on the machine comsat is running on.

The exact property we are looking for here is whether a user can write arbitrary data to the `/etc/passwd` password file. If that is the case then that user can easily set a new password that is known to him for the root user. Two conditions exist under which this is possible:

- The user has write permission on `/etc/passwd`. This is a trivial solution.

- the user has write permission on `/etc/utmp`[2] This file stores user login information, including the user's terminal. By setting the root's terminal to `/etc/passwd` and sending a mail to root, any user can manage to set a new root password.

A simple model of the system that only allows to find the first solution was available to the authors. Nevertheless it contains the typical features of an extended finite automaton, while not overly drawing attention to the complexity of the problem at hand, to serve as a good proof of concept for CHR in a tabled context.

Constraints perform three tasks in this application:

- They record the conditions under which state transitions in the model can be taken, and as such specify the conditions under which the system is vulnerable to attack by a user.
- They allow to rule out the uninteresting case that the user is root.
- They enable avoidance of impossible transitions in the model through failure because of unsatisfiable constraints.

The previous implementation of the constraints used an explicit list-based constraint store. From time to time consistency checks, simplification and projection were performed on this store.

It was easily replaced by 2 constraints, `neq/2` for user inequality and `exists/2` for the existence of a file in a file system.

neq(User,Name) \ neq(User,Name) $\Longleftrightarrow$ true.
exists(File,FileSystem) \ exists(File,FileSystem) $\Longleftrightarrow$ true.
exists(file(Name,Permissions,Data),FS) $\Longleftrightarrow$ lookup(Name,FS,P,D) | Permissions = P, Data = D.

project(User,FS) \ neq(AUser,FS) $\Longleftrightarrow$ User \ == AUser | true.
project(User,FS) \ exists(File,AFS) $\Longleftrightarrow$ AFS \ == FS | true.

lookup(Name,FS,Permissions,Data) :- member(file(Name,Permissions,Data),FS).

The `reach/2` predicate was transformed to do full constraint abstraction and the above described projection onto variables of interest.

With this CHR implementation we have much more confidence in the scheduling of projection, simplification and satisfiability checking. The implementation is much more compact and readable as well. The program runs in less than a millisecond.

4.2 Model Checking of Real Time Systems

The problem looked at here is model checking for timed automata. See [16] for an overview of different techniques.

Timed automata are automata with a finite set of clocks that can take continuous values between 0 and ∞. All clocks advance synchronously. Transitions between states can have upper and lower bound constraints on clock values. In addition, clocks can be reset on a transition. In a state clocks can delay for any amount of time before taking a next transition.

[2] The actual location of this file may vary between different Unix systems.

The constraint solving subproblem consists of determining the upper and lower bounds on all clocks for a certain transition given:

- the lower bound of the clocks at the start state of the transition
- the constraints on the clocks for the transition
- the maximum pairwise distances between clocks determined by resets

The previous implementation in the XMC toolkit (see [12]) used distance bound matrices (DBMs) to represent constraint stores. The necessary matrix manipulations were provided to pose constraints, determine the canonical form, reset clocks and delay.

We have replaced the DBM implementation with CHR constraints:

- `lowerbound(N,O,C)` with `N` a number, `O` either $\leq$ or $<$ and C a clock. This expresses the constraint that $n \leq c$ or $n < c$.
- `upperbound(N,O,C)` with `N` a number, `O` either $\geq$ or $>$ and C a clock. This expresses the constraint that $c \leq n$ or $c < n$.
- `diff(N,O,C1,C2)` with `N` a number, `O` either $\geq$ or $>$ and both C1 and C2 clocks. This is used to model the distance constraints. Its meaning is that $n \geq c_1 - c_2$ or $n > c_1 - c_2$.

Clock resets and delays have been implemented as operations on a list representation of the constraint store. For the actual constraint solving constraints on the clocks 24 CHR rules have been implemented, of which the following 3 are a sample :

upperbound(X,$\leq$,N1) \ upperbound(X,$\leq$,N2) $\Longleftrightarrow$ N1 $\leq$ N2 | true.

upperbound(X,$\leq$,N1), lowerbound(X,$\geq$,N2) $\Longrightarrow$ N1 $\geq$ N2.

dist(X,Y,$\leq$,D), upperbound(Y,$\leq$,N) $\Longrightarrow$ M is N + D, upperbound(X,$\leq$,M).

In this application we used tabling on two levels: `reach/2` as well as `edge/2` transitions.

In contrast to the previous application and the general approach discussed in Section 3 no constraint call abstraction is performed. Instead the constraint store is passed around explicitly when not doing any constraint solving. As no auxiliary variables are introduced, no projection is applied either. To take care of clocks delaying in states before taking a transition, upper bound constraints on them are removed in the explicit representation of the constraint store. For the solving of the constraint on an edge transition the explicit constraint store is converted to the usual global CHR constraint store and afterwards the other way around.

Instead of performing constraint abstraction in the tabled predicates, clock name abstraction is performed: i.e. clock names are replaced by variables. This allows answer reuse for identical call constraint stores modulo clock names. This optimization is quite useful in the case of parallelization of multiple identical automata. The clock names will be different for different instances of the automaton, but states and constraints are otherwise identical. The original DBM implementation is not easily capable of this optimization as rows and clocks implicitly correspond with clock names and reordering of rows and columns on other criteria would be non-trivial compared to the near canonicalization performed on the CHR list representation.

The preliminary results of this implementation are that performance equal to that of the DBM implementation has been achieved. The code however is much more concise and more confidence in the correctness of the implementation has been achieved as it is much closer to the semantics of the problem domain than the mapping onto DBM matrix manipulations. Moreover, future work on this application will explore high level optimizations enabled by the CHR implementation and semantical extensions of tabled automata such as different kinds of constraints on clocks.

5 Conclusion and Future Work

In this paper we have shown that it is possible to integrate the committed choice bottom-up execution of CHRs with the tabled top-down execution of XSB. In particular the issues related to the consistency of the global CHR store and tables have been established and solutions have been formulated for call abstraction, tabling constraint stores, answer projection, answer combination (e.g. for optimization), and answer entailment checking.

Furthermore CHR with tabling has proven a powerful combination: both CHR and tabled XSB relieve the programmer of the complicated underlying scheduling mechanism behind the scenes and put the focus on higher level semantics.

Model checking provides a rich application field. The combination of CHR and XSB extends the conciseness of ordinary model checking systems to those with constraints. Indeed the next step in the search for applications is to explore more expressive models than are currently viable with traditional approaches: the flexible nature of CHR makes it easy to experiment with various types of custom constraints.

Competitive performance has been observed in the model checking domain. The high level nature of the CHR implementation has revealed some optimizations not apparent or feasible to other implementations. These and more general performance aspects will be explored in future work.

As mentioned earlier in Section 3.5, pattern equality testing of constraint stores remains a challenge. The efficiency and accuracy of the used algorithm may have a considerable impact on the overall runtime of particular applications.

Other aspects of tabling constraints that have not been touched in this paper are how to implement partial abstraction and the implications for variant and subsumption based tabling. Partial abstraction and subsumption are closely related. The former transforms a call into a more general call while the latter looks for answers to more general calls, but if none are available still executes the actual call.

It is also worth mentioning that an XSB release with the presented CHR system will soon be publicly available (see `http://xsb.sf.net`).

Acknowledgements

We would like to thank Giridhar Pemmasani and Beata Sarna-Starosta for allowing and assisting us to rework their model checking systems with CHR constraint solvers.

Tom Schrijvers is a research assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

References

[1] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[2] B. Cui and D. S. Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.

[3] B. Cui and D. S. Warren. Attributed Variables in XSB. In I. Dutra, V. S. Costa, G. Gupta, E. Pontelli, M. Carro, and P. Kacsuk, editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier, 2000.

[4] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.

[5] B. Demoen, M. G. de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An Overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, 1999.

[6] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[7] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled Resolution + Constraints: A Recipe for Model Checking Real-Time Systems. In *IEEE Real Time Systems Symposium (RTSS)*, Orlando, Florida, November 2000.

[8] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, March 1995.

[9] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[10] C. Holzbaur and T. Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.

[11] M. Mukund, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. Verma. Symbolic Bisimulation using Tabled Constraint Logic Programming. In *International Workshop on Tabulation in Parsing and Deduction (TAPD)*, Vigo, Spain, September 2000.

[12] G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient Model Checking of Real Time Systems Using Tabled Logic Programming and Constraints. In *International Conference on Logic Programming (ICLP)*, Lecture Notes in Computer Science, Copenhagen, Denmark, July 2002. Springer.

[13] C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A Logic-Programming-Based Verification Toolset. In

Twelfth International Conference on Computer Aided Verification (CAV), volume 1855 of *Lecture Notes in Computer Science*, pages 576–580, Chicago, Illinois, July 2000. Springer.

[14] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. The XSB Programmer's Manual: version 2.5, vols. 1 and 2, 2001.

[15] B. Sarna-Starosta and C. Ramakrishnan. Constraint-Based Model Checking of Data-Independent Systems. In *5th International Conference on Formal Engineering Methods*, 2003.

[16] S. Yovine. Model Checking Timed Automata. In *European Educational Forum: School on Embedded Systems*, pages 114–152, 1996.

Simplifying Dynamic Programming via Tabling

Hai-Feng Guo
Computer Science Department, University of Nebraska at Omaha
Omaha, NE 68182-0500, USA
Email: haifengguo@mail.unomaha.edu

Gopal Gupta
Computer Science Department, University of Texas at Dallas
Richardson, TX 75083-0688 USA
Email: gupta@utdallas.edu

Abstract

In the dynamic programming paradigm the value of an optimal solution is recursively defined in terms of optimal solutions to subproblems. This definition can be very tricky and error-prone to specify. This paper presents a novel, elegant method based on tabled logic programming that simplifies the specification of such dynamic programming solutions. Our method introduces a new mode declaration for tabled predicates. The arguments of each tabled predicate are divided into indexed and non-indexed ones so that tabled predicates can be regarded as functions: indexed arguments represent input values and non-indexed arguments represent output values. The non-indexed arguments in a tabled predicate can be further declared to be *aggregate*, e.g. the minimum, so that while generating answers, the global *table* will dynamically maintain the smallest value for that argument. This mode-declaration scheme, coupled with recursion, provides a considerably easy-to-use method for dynamic programming: there is no need to define the value of an optimal solution recursively, instead, defining a general solution suffices. The optimal value as well as its corresponding concrete solution can be derived implicitly and automatically using tabled logic programming systems. Experimental results are shown to indicate that the mode declaration improves both time and space performances in solving dynamic programming problems on tabled LP systems.

1 Introduction

Tabled logic programming (TLP) systems [2, 3, 5, 6] have been put to many innovative uses, such as model checking [7] and non-monotonic reasoning [10], due to their highly declarative nature and efficiency. A tabled logic programming system can be thought of as an engine for efficiently computing fixed points, which is critical for many practical applications. A TLP system is essential for extending traditional LP system (e.g., Prolog) with tabled resolutions. The main advantages of tabled resolution are that a TLP system terminates more often by computing fixed points, avoids redundant computation by memoing the computed answers, and keeps the declarative and procedural semantics consistent for pure logic programs.

The main idea of tabled resolution is never to compute the same call twice. Answers to certain calls are recorded in a global *memo table* (heretofore referred to as a *table*), so that whenever the same call is encountered later, the tabled answers are retrieved and used instead of being recomputed. This avoidance of recomputation not only gains better efficiency, more importantly, it also gets rid of many infinite loops, which often occur due to static computation strategies (e.g. SLD resolution [1]) adopted in traditional logic programming systems.

Dynamic programming algorithms are particularly appropriate for implementation with tabled logic programming [9]. Dynamic programming is typically used for solving optimization problems. It is a general recursive strategy in which optimal solution to a problem is defined in terms of optimal solutions to its subproblems. Dynamic programming, thus, recursively reduces the solution to a problem to repetitively solving its subproblems. Therefore, for computational efficiency it is essential that a given subproblem is solved only once instead of multiple times. From this standpoint, tabled logic programming *dynamically* incorporates the dynamic programming strategy [9] in the logic programming paradigm. TLP systems provide implicit tabulation scheme for dynamic programming, ensuring that subproblems are evaluated only once.

In spite of the assistance of tabled resolution, solving practical problems with dynamic programming is still not a trivial task. The main step of dynamic programming paradigm is to define the value of an optimal solution recursively in terms of the optimal solutions to subproblems. This definition could be very tricky and error-prone. As the most widely used TLP system, XSB provides table aggregate predicates [2, 10], such as `bagMin/2` and `bagMax/2`, to find the minimal or maximal value from tabled answers respectively. Those predicates are helpful in finding the optimal solutions, and therefore in implementing dynamic programming algorithms. However, users still have to define optimal solutions explicitly, that is, specify how the optimal value of a problem is recursively defined in terms of the optimal values of its subproblems. Furthermore, the aggregate predicates require the TLP system to collect all possible values, whether optimal or non-optimal, into the memo table, which could dramatically increase the table space needed.

Another important issue in dynamic programming is that once the optimal value is found for a problem, the concrete solution leading to that optimal value needs to be constructed. This requires that each computed value be associated with some evidence (or explanation [11]) for solution construction. In the tabled logic programming formulation, an extra argument is added to the tabled predicates in which a representation of the explanation is conveniently built. Unfortunately, to put explanation as an extra tabled predicate argument results in recording of the explanation as part of the answers to tabled calls. This can dramatically increase the size of the global table space because there can be many explanations for a single answer in the original program. Similar issues are raised in [9] on generating parse-trees: determining whether there is a parse-tree can be done in time cubic on the length of the string (worst case) whereas the number of parse trees may be exponential. Therefore, from a complexity standpoint, use of TLP for dynamic programming has certain negative aspects.

This paper presents a novel declarative method based on the tabled logic programming paradigm for simplifying dynamic programming solutions to problems. The method introduces a new mode declaration for tabled predicates. The mode declaration classifies arguments of a tabled predicate as indexed or non-indexed. Each non-indexed argument can be thought of as a function value uniquely determined by indexed arguments. The tabled logic programming system is optimized to perform variant checking based only on the indexed arguments. This new declaration for tabled predicates and modified procedure for variant checking makes it easier to collect a single associated explanation for a tabled answer, e.g., a concrete solution for an optimal value in dynamic programming paradigm, even though, in principle there may exist a lot of explanations for the same tabled answer. The collected explanation can be shown very concisely without involving any self-dependency among tabled subgoals.

The mode declaration can further extend one of the non-indexed arguments to be an aggregated value, e.g. the minimum function, so that the global *table* will record answers with the value of that argument appropriately aggregated. Thus, in the case of the minimum function, a tabled answer can be dynamically replaced by a new one with a smaller value during the computation. This mode declaration is essential for obtaining the optimal solution from a general specification of the dynamic programming solution.

This new mode-declaration scheme, coupled with recursion, provides an attractive platform for making dynamic programming simpler: there is no need to define the value of an optimal solution recursively, instead, defining the value of a general solution is enough. The optimal value, as well as its associated solution, will be computed implicitly and automatically in a tabled logic programming system that uses the new mode declaration and modified variant checking. Thus, dynamic programming problems are solved more elegantly.

The rest of the paper is organized as follows: Section 2 introduces tabled logic programming, in particular, implemented using *dynamic reordering of alternatives (DRA) [5]*. DRA is described in subsection 2.1, followed by the typical tabled logic programming based approach for dynamic programming in subsection 2.2. Section 3 presents our new annotation for declaring tabled goals, followed by a detailed demonstration of how dynamic programming can benefit from this new scheme. Section 4 presents the running performance on some dynamic programming benchmarks. Finally, section 5 presents our conclusions.

2 Tabled Logic Programming (TLP)

Traditional logic programming systems (e.g., Prolog) use SLD resolution [1] with the following *computation strategy*: subgoals of a resolvent are solved from left to right and clauses that match a subgoal are applied in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. That is, given any static computation strategy, one can always produce a program in which no answers can be found due to non-termination even though some answers may logically follow from the program. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

Tabled logic programming eliminates such infinite loops by extending logic programming with tabled resolution. The main idea is to memorize the answers to some calls and use the memorized answers to resolve subsequent variant calls. Tabled resolution adopts a dynamic computation strategy while resolving subgoals in the current resolvent against matched program clauses or tabled answers. It keeps track of the nature and type of the subgoals; if the subgoal in the current resolvent is a variant of a former tabled call, tabled answers are used to resolve the subgoal; otherwise, program clauses are used following SLD resolution.

In a tabled logic programming system, only tabled predicates are resolved using tabled resolution. Tabled predicates are explicitly declared as `:- table p/n.`, where `p` is a predicate name and `n` is its arity. A global data structure *table* is introduced to memorize the answers of any subgoals to tabled predicates, and to avoid any recomputation.

We use *dynamic reordering of alternatives* (DRA) resolution [5] throughout this paper as an example of tabled resolution. Other tabled resolutions, including SLG [2], SLDT [3], etc. perform similarly toward the computation of a fixed point.

2.1 Dynamic Reordering of Alternatives (DRA)

The DRA resolution computes a fixed point in a very similar way as bottom-up execution of logic programs [1]. Its main idea is to dynamically identify *looping alternatives* from the program clauses, and then repetitively apply those alternatives until no more answers can be found. A looping alternative refers to a clause that matches a tabled call and will lead to a resolvent containing a recursive variant call.

Program 2.1 *Consider the following tabled logic program defining a reachability relation:*

```
:- table reach/2.
reach(X, Y) :- reach(X, Z), arc(Z, Y).      (1)
reach(X, Y) :- arc(X, Y).                   (2)
arc(a, b).     arc(a, c).     arc(b, a).
:- reach(a, X).
```

As shown in Figure 1, the computation of `reach(a,X)` is divided into three stages: *normal*, *looping* and *complete*. The purpose of the normal stage is to find all the looping alternatives (the clause (1) leading to a variant subgoal `reach(a,Z)`) and record all the answers generated from the non-looping alternatives (the clause (2)) into the table. The `new_answer` label indicates that the new answer generated from that successful path should be added into the table. Then, in the looping stage only the looping alternative (clause (1)) is performed repeatedly to consume new tabled answers until a fixed point is reached, that is, no more answers for `reach(a,X)` can be found. Afterwards, the complete stage is reached. As a result, the query `:- reach(a,X)` returns a complete answer set `X=b`, `X=c` and `X=a`, albeit the predicate is defined left-recursively.

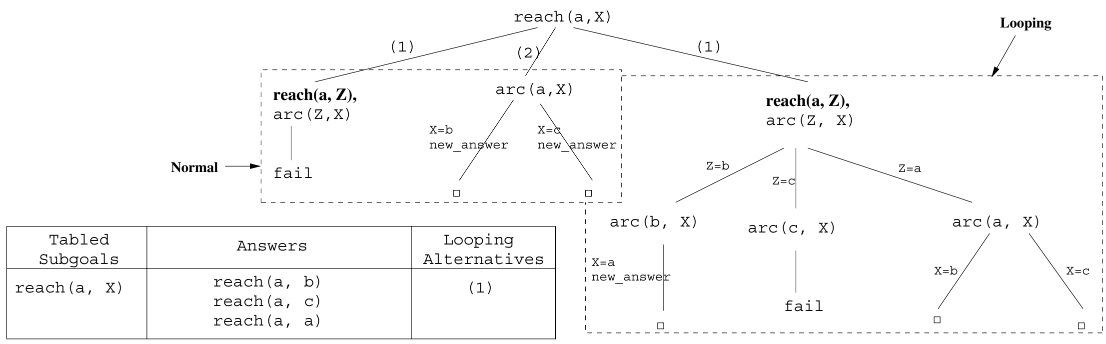


Figure 1: DRA Resolution 2.1

2.2 Dynamic Programming with TLP

We use the *matrix-chain multiplication* problem [12] as an example to illustrate how tabled logic programming can be adopted for solving dynamic programming. A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Thus, the matrix-chain multiplication problem can be stated as follows (detailed description of this problem can be found in any major algorithm textbook covering dynamic programming):

Problem 2.1 *Given a chain* $\langle A_1, A2, ..., A_n\rangle$ *of* n *matrices, where for* $i = 1, 2, ..., n$, *matrix* A_i *has dimension* $p_{i-1} \times p_i$, *fully parenthesize the product* $A_1A_2...A_n$ *in a way that minimizes the number of scalar multiplications.*

To solve this problem by dynamic programming, we need to define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, which denotes a sub-chain of matrices $A_i A_{i+1} ... A_j$ for $1 \leq i \leq j \leq n$. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_{i..j}$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

A tabled Prolog coding is given in Program 2.2 to solve the matrix-chain multiplication problem. The predicate `scalar_cost(PL, V, P0, Pn)` is tabled, where `PL`, `P0` and `Pn` are given by the user to represent the dimension sequence $[p_0, p_1, ..., p_n]$, the first dimension p_0 and the last dimension p_n, respectively, and `V` is the minimum cost of scalar multiplications to multiply $A_{1..n}$; the built-in predicate `findall(X,G,L)` is used to find all the instances of `X` as a list `L` such that each instance satisfies the goal `G`; the predicate `break(PL, PL1, PL2, Pk)` is used to split the dimension sequence at the point of `Pk` into two parts to simulate the parenthesization; and the predicate `list_min(L, V)` finds a minimum number `V` from a given list `L`.

Program 2.2 *A tabled logic program for matrix-chain multiplication problems:*

```
:- table scalar_cost/4.
scalar_cost([P1, P2], 0, P1, P2).
scalar_cost([P1, P2, P3 | Pr], V, P1, Pn) :-
    findall(V, ( break([P1, P2, P3 | Pr], PL1, PL2, Pk),
                 scalar_cost(PL1, V1, P1, Pk),
                 scalar_cost(PL2, V2, Pk, Pn),
                 V is V1 + V2 + P1 * Pk * Pn ), VL),
    list_min(VL, V).
break([P1, P2, P3], [P1, P2], [P2, P3], P2).
break([P1, P2, P3, P4 | Pr], [P1, P2], [P2, P3, P4 | Pr], P2).
break([P1, P2, P3, P4 | Pr], [P1 | L1], L2, Pk) :-
    break([P2, P3, P4 | Pr], L1, L2, Pk).
```

Consider a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. We can give a query `:- scalar_cost([10, 100, 5, 50], V, 10, 50)` to find the minimum value of its scalar multiplications. As a result, `V` is instantiated to 7500, corresponding to the optimal parenthesization $((A_1 A_2) A_3)$.

Program 2.2 shows that the programmer has to find the optimal value by comparing all possible multiplication costs explicitly. In fact, for a general optimization problem, the definition of an optimal solution could be quite complicated due to heterogeneous solution construction. Then, comparing all possible solutions explicitly to find the optimal one could be very tricky and error-prone. In this paper we present a simple method to separate the task of finding the optimal solution from the task of specifying the general dynamic programming formulation. Using our method, the programmer is only required to define *what* a general solution is, while searching for the optimal solution is left to the TLP system.

The next thing we are interested in is finding the actual parenthesization (explanation) that led to the optimal answer. Of course, the above program is of no help, since it only finds the optimal value for the number of scalar multiplications. A standard method [11] to construct explanation in logic programming is to add an extra argument to tabled predicates for the explanation. However, this extra argument results in recording explanation as part of the answers to tabled calls, which can dramatically increase the size of global table space. Consider the program for computing reachability again, we can introduce a new transformed tabled predicate `reach/3` as shown in Program 2.3, where the third argument `E` is used to generate the path from `X` to `Y`. Obviously, there are infinite number of paths from `a` to any node due to the cycle between `a` and `b`. Therefore, from this standpoint of computational complexity, tabling predicates has certain drawbacks. In this paper we show how this drawback can be removed.

Program 2.3 *A tabled logic program defining a reachability relation predicate with path information as an extra argument:*

```
:- table reach/3.
reach(X, Y, E) :- reach(X, Z, E1), arc(Z, Y, E2), append(E1, E2, E).
reach(X, Y, E) :- arc(X, Y, E).
arc(a, b, [(a,b)]).    arc(a, c, [(a,c)]).    arc(b, a, [(b,c)]).
:- reach(a, Y, E).
```

Similar problems have been studied on justification in [13, 8]. One reasonable solution is presented in [8] by asserting the first evidence into a dynamic database for each tabled answer. However, the evidence has to be organized as segments indexed by each tabled answer. That is, an extra procedure is required to construct the full evidence.

3 A Declarative Method

In this section, we present a new method by introducing a special mode declaration for tabled predicates. The mode declaration is used to classify arguments as indexed or non-indexed for each tabled predicate. Only indexed arguments in a tabled predicate are used for variant checking.

Variant checking is a crucial operation for tabled resolution as it leads to avoidance of non-termination. It is used to differentiate both tabled goals and their answers. While computing the answers to a tabled goal p with tabled resolution, if another tabled subgoal q is encountered, the decision regarding whether to consume tabled answers or to try program clauses depends on the result of variant checking. If q is a variant of p, the variant subgoal q will be resolved by unifying it with tabled answers, otherwise, traditional Prolog resolution is adopted for q. Additionally, when an answer to a tabled goal is generated, variant checking is used to check whether the generated answer is variant of an answer that is already recorded in the table. If so, the table is not changed; this step is crucial in ensuring that a fixed point is reached.

Notice that the new method can also be applied on other tabled resolutions, such as SLG [14] and SLDT [3], since essentially only variant checking is modified.

3.1 Mode Declaration for Evidence Construction

The new mode declaration for tabled predicates can be described in the form of

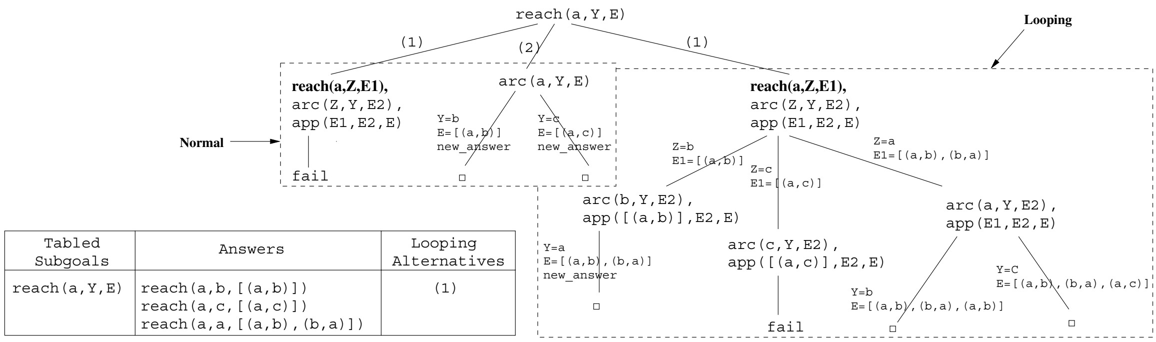
$$\texttt{:- table\_mode } p(a_1, ..., a_n)\texttt{.}$$

where p is a tabled predicate name, $n \geq 0$, and each a_i has one of the following forms:

$+$ denotes that this indexed argument is used for variant checking;

$-$ denotes that this non-indexed argument is not used for variant checking.

Consider the reachability example again. Suppose we declare the mode as "`:- table_mode` *reach*$(+,+,-)$"; this means that the first two arguments of the predicate `reach/3` are used for variant checking. The new computation of the query `reach(a,Y,E)` is shown in Figure 2. Since only the first two arguments of `reach/3` are used for variant checking, the last two answers "`Y=b, E=[(a,b),(b,a),(a,b)]`" and "`Y=c, E=[(a,b),(b,a),(a,c)]`", shown on the rightmost two sub-branches, are variant answers to "`Y=b, E=[(a,b)]`" and "`Y=c, E=[(a,c)]`" respectively. Therefore, no new answers are added into the table at those points. The computation is then terminated properly with three answers. As a result, each reachable node from `a` has a simple path.



Tabled Subgoals	Answers	Looping Alternatives
reach(a,Y,E)	reach(a,b,[(a,b)]) reach(a,c,[(a,c)]) reach(a,a,[(a,b),(b,a)])	(1)

Figure 2: DRA Resolution with Table Mode Declaration

The mode directive `tabled_mode` makes it very easy and efficient to extract explanation for tabled predicates. In fact, our strategy of ignoring the explanation argument during variant checking results in only the first explanation for each tabled answer being recorded. Subsequent explanations are filtered by our modified variant checking scheme. This feature ensures that those generated explanations are concise and that cyclic explanations are guaranteed to be absent. For the reachability instance shown in Figure 2, each returned path is simple such that all arcs are distinct.

Essentially, if we regard a tabled predicate as a function, then all the non-indexed arguments are uniquely defined by the instances of indexed arguments. For the previous example, the third argument of `reach/3` returns a single path depending on the first two arguments. Therefore, variant checking should be done w.r.t. only indexed arguments during tabled resolution. Indexed arguments in a tabled predicate can also be declared with the mode '`*`' if they are always bound before a call to the tabled predicate is invoked. In this case, even though a tabled call may have many answers, they share the same input arguments. Therefore, for each tabled call, only one copy of the input arguments need be stored. From these viewpoints, the mode declaration makes tabled resolution more efficient and flexible.

3.2 Aggregate Declaration for Making Dynamic Programming Easier

The mode directive `table_mode` can be further extended to associate a non-indexed argument of a tabled predicate with some optimum constraint. Currently, a non-indexed argument for each tabled answer only records the very first instance. This "very first" property can actually be generalized to any optimum, e.g. the minimum value, in which case the global table will record answers with the value of that argument as small as possible. That is, a tabled answer can be dynamically replaced by a new one with smaller value during the computation. In general, given a tabled call `p(`$\mathtt{I}_1$`, `$\mathtt{I}_2$`, ..., `$\mathtt{I}_n$`, NI)` where $\mathtt{I}_i$ for $1 \le i \le n$ are indexed arguments and `NI` is a single non-indexed argument, let `p(`$\mathtt{i}_1$`, `$\mathtt{i}_2$`, ..., `$\mathtt{i}_n$`, u)` be the entry for predicate `p` (with arity `n+1`) in the table.

Suppose, a new solution is found during tabled execution represented by `p(i`$_1$`, i`$_2$`, ..., i`$_n$`, v)`, then, in general, the user can define an aggregation predicate f such that the new value of the non-indexed argument is updated to `w` such that `f(u,v,w)` holds. By default, `f` is defined as `f(X,_,X)` (the value of the non-indexed argument is set to the first one found). For dynamic programming applications, `f` is set to the *min* predicate as follows:

`f(X,Y,Z) :- min(X,Y,Z).`

This can easily be generalized to the case where multiple non-indexed arguments are present.

Currently, we allow `min` and `max` as the only aggregation predicates. These are specified via special mode declarations. Two new modes are added in the directive `table_mode` to declare the aggregation operation to be used for tabled predicates; both modes also imply that the arguments are non-indexed.

0 denotes that this argument is a minimum;

9 denotes that this argument is a maximum.

The aggregation declaration can be used to make control of execution during dynamic programming implicit, making the specification of dynamic programming problems more declarative and elegant. For the matrix-chain multiplication, instead of defining the cost of an optimal solution, we only need to specify what the cost for a general solution is. Let $m[i,j]$ be the number of scalar multiplications needed to compute the matrix $A_{i..j}$ for $1 \le i \le j \le n$, where n is the total number of matrices. The recursive definition for the cost of parenthesizing $A_{i..j}$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ m[i,k] + m[k+1,j] + p_{i-1}p_k p_j & \text{if } i < j. \end{cases}$$

Program 3.1 *A tabled logic program with optimum mode declaration for matrix-chain multiplication problems:*

```
:- table scalar_cost/4.
:- table_mode scalar_cost(+, 0, -, -).
scalar_cost([P1, P2], 0, P1, P2).
scalar_cost([P1, P2, P3 | Pr], V, P1, Pn) :-
    break([P1, P2, P3 | Pr], PL1, PL2, Pk),
    scalar_cost(PL1, V1, P1, Pk),
    scalar_cost(PL2, V2, Pk, Pn),
    V is V1 + V2 + P1 * Pk * Pn.
```

The mode declaration `scalar_cost(+,0,-,-)` means that only the first argument (the list of matrix dimensions) is used for variant checking when an answer is generated, and a minimum value is expected for the second argument (the cost of scalar multiplication). Arguments with different modes are tested in the following order during variant checking of a recently generated answer: (1) the indexed argument with '`+`' mode has the highest priority to be first checked to identify whether it is a new answer; if that is the case, a new tabled entry is required to record the answer; otherwise a tabled answer with the same indexed argument is found; (2) this tabled answer is then compared with the recently generated one w.r.t the argument with the optimum mode '`0`'; if the new answer has a smaller value on the optimum argument, then a replacement over the tabled answer is required such that the tabled answer keeps the minimum value as expected for this argument.

Figure 3 shows the recursion tree produced by the query

```
:- scalar_cost([10,100,5,50],V,10,50).
```

Consider the tabled call `scalar_cost([10,100,5,50],V,10,50)`. Its first tabled answer has `V=75000`. However, when the second answer `V=7500` is computed, it will automatically replace the previous answer following the declared optimum mode. Thus, there is at most one instance of `scalar_cost([10,100,5,50],V,10,50)` that exists in the table at any point in time, and it represents the optimal value computed up to that point.

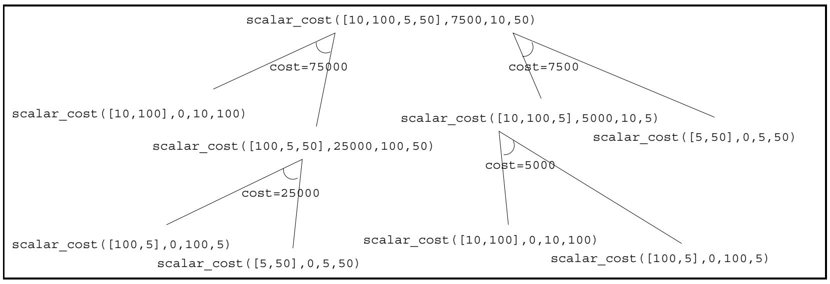


Figure 3: The recursion tree for the computation of `scalar_cost([10,100,5,50],V,10,50)`

As long as the tabled Prolog engine is set to compute the fixed point semantics for logic programs with bounded term depth, the optimal value for the dynamic programming problem under consideration will always be found. Intuitively, given a tabled call $\mathcal{C}$, the DRA resolution first finds all the answers for $\mathcal{C}$ using clauses not containing variant calls. Once this set of answers is computed and tabled, it is treated as a set of facts, and used for computing rest of the answers from the clauses leading to variant calls (looping alternatives). Whenever an answer to $\mathcal{C}$ is generated, it will be selectively added to the table either as a new entry or as a replacement based on the defined mode of the corresponding predicate. The process stops when no new answers can be computed via the looping alternatives, i.e., a fixed point is reached. In this regard, with the assistance of mode declaration and tabled resolution, the computation of program clauses only defining general solutions will still produce the optimal solution.

3.3 Dynamic Programming with Evidence Construction

To make the matrix-chain multiplication problem complete, we need to construct an optimal parenthesization solution corresponding to the minimal cost of scalar multiplication. This construction can be achieved with the strategy described in Section 3.1, by introducing an extra non-indexed argument whose instantiation becomes the solution. The complete tabled logic program is shown below:

Program 3.2 *A tabled logic program for the complete matrix-chain multiplication problem:*

```
:- table scalar_cost_evid/5.
:- table_mode scalar_cost_evid(+, 0, -, -, -).
scalar_cost_evid([P1, P2], 0, P1, P2, (P1,P2)).
scalar_cost_evid([P1, P2, P3 | Pr], V, P1, Pn, (E1*E2)) :-
    break([P1, P2, P3 | Pr], PL1, PL2, Pk),
    scalar_cost_evid(PL1, V1, P1, Pk, E1),
    scalar_cost_evid(PL2, V2, Pk, Pn, E2),
    V is V1 + V2 + P1 * Pk * Pn.
```

4 Experimental Results

The mode declaration scheme has been implemented in the authors' TALS [5] system, a tabled Prolog system implemented on the top of the WAM engine of the commercial ALS Prolog engine [15]. No change is required to the DRA resolution mechanism; therefore, the same idea can also be applied to other tabled Prolog systems.

Two major changes to the global data structure *table* are needed to support mode declarations. First, each table predicate is associated with a new item *mode*, which is represented as a bit string. The default mode for each argument in a table predicate is '`-`'. Second, the answers to a tabled subgoal are selectively tabled depending on its mode declaration. The input arguments with the mode '`*`' in each tabled subgoal are only recorded once because the input arguments are bound before the subgoal is invoked, and therefore same for all its answers. This optimization leads to big improvements on both time and space system performance.

Another important implementation issue is the replacement of tabled answers. In the current TALS system, if the tabled subgoal only involves numerals as arguments, then the tabled answer will be completely replaced if necessary. If the arguments involve structures, however, then the answer will be updated by a link to the new answer. Space taken up by the old answer has to be recovered by garbage collection (the ALS Prolog's garbage collector has not yet been extended by us to include table space garbage recovery). As a result, if arguments of tabled predicates are bound to structures, more table space is used up.

Our experimental benchmarks include five typical dynamic programming examples. `matrix` is the matrix-chain multiplication problem; `lcs` is longest common subsequence problem; `obst` finds an optimal binary search tree; `apsp` finds the shortest paths for all pairs of nodes; and `knap` is the knapsack problem. All tests were performed on an Intel Pentium 4 Mobile CPU 1.8GHz machine with 512M RAM running RedHat Linux 9.0.

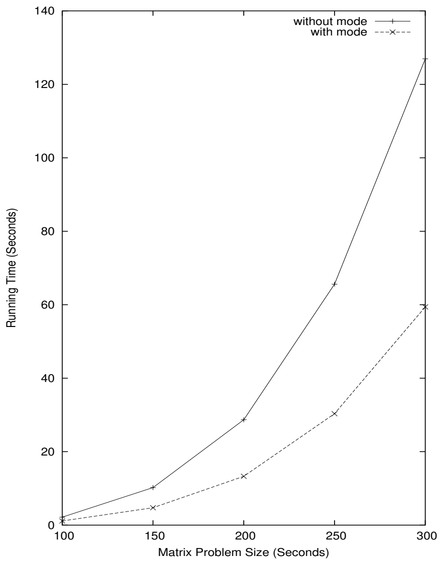
	without evidence construction					with evidence construction				
Benchmark	**matrix**	**lcs**	**obst**	**apsp**	**knap**	**matrix**	**lcs**	**obst**	**apsp**	**knap**
without mode	2.18	0.94	0.90	4.17	54.59	3.09	5.93	11.69	6.70	140.46
with mode	1.14	0.43	0.32	2.90	40.64	2.27	0.67	0.73	3.10	41.77

Table 1: Running time performance comparison (Seconds)

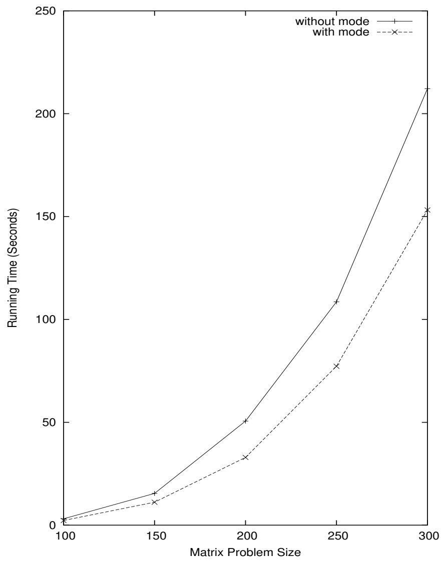
Table 1 compares the running time performance between the programs with and without mode declaration. The first group of benchmarks are programs only seeking the optimal values without evidence construction, while the second group are programs for the same dynamic programming problems with evidence construction. The experimental data indicates that the programs with mode declaration run 1.34 to 16.0 times faster than the corresponding programs without mode declaration.

Figure 4 shows the timing information against different input sizes for matrix-chain multiplication problems. Notice that the numbers on X-axis represent the total number of matrices to be multiplied, and the numbers on Y-axis represent the running time with seconds. Whether without evidence construction (Figure 4(a) or with evidence construction (Figure 4(b)), the graphs indicate that the timings of the programs with mode are consistently better than those without mode declaration.

Additionally, we compare the running space performance between the programs with and without mode declaration in Table 2. For benchmarks without evidence construction, our experiments



(a) Without evidence



(b) With evidence

Figure 4: Time Performance of Matrix-chain Multiplication

	without evidence construction					with evidence construction				
Benchmark	**matrix**	**lcs**	**obst**	**apsp**	**knap**	**matrix**	**lcs**	**obst**	**apsp**	**knap**
without mode	4.98	78.75	2.65	20.17	222.65	9.22	92.99	4.44	29.90	399.95
with mode	0.57	23.44	0.25	14.60	14.86	11.64	44.24	17.46	21.39	305.19

Table 2: Running space comparison (Megabytes)

indicate that with mode declaration, space requirement is 1.4 to 15.0 times less compared to without mode declaration. With evidence construction included, space performance can be better or worse depending on the problems. For the `matrix` and `obst` problems trying to find the optimal binary tree structure, the programs without mode explicitly generate all possible answers and then table the optimal one, while the programs with mode implicitly generate all possible answers and selectively table the better answers until the optimal one is found. In the latter case, some non-optimal answers may be recorded in the table, unseen by the user; if the optimal answer happens to be the first tabled answer, then no other un-optimal answers will be recorded. This is the reason why the benchmarks `matrix` and `obst` (with evidence construction) with mode declaration take more space than those without mode, as shown in Table 2.

5 Conclusion

A new mode declaration for tabled predicates is introduced in TLP systems to aggregate information dynamically recorded in the table. The mode declaration classifies arguments of tabled predicates as either indexed or non-indexed. As a result, (i) a tabled predicate can be regarded as a function in which non-indexed arguments (outputs) are uniquely defined by the indexed arguments (inputs); (ii) concise explanation for tabled answers can be easily constructed in a non-indexed argument;

(iii) the efficiency of tabled resolution is improved since only indexed arguments are involved in variant checking; and (iv) the non-indexed arguments can be further qualified with an aggregate mode such that an optimal value can be sought without explicit coding of the comparison.

This new mode declaration scheme, coupled with recursion, provides an elegant method for solving dynamic programming problems: there is no need to define the value of an optimal solution recursively, instead, defining the value of a general solution is enough. The optimal value, as well as its associated solution, is obtained automatically by the TLP systems. This new scheme has been implemented in the authors' TALS system with very encouraging results.

References

[1] J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.

[2] XSB system. http://xsb.sourceforge.net

[3] Neng-Fa Zhou, Y. Shen, L. Yuan, and J. You: Implementation of a Linear Tabling Mechanism. In Proceedings of *Practical Aspects of Declarative Languages (PADL)*, 2000.

[4] I.V. Ramakrishnan, P. Rao, K.F. Sagonas, T. Swift, D.S. Warren: Efficient table access mechanisms for logic programs. *Journal of Logic Programming*, 38(1):31-54, Jan. 1999.

[5] Hai-Feng Guo and Gopal Gupta: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In Proceedings of *International Conference on Logic Programming (ICLP)*, pages 181–196, 2001.

[6] R. Rocha, F. Silva, and V. S. Costa: On a Tabling Engine That Can Exploit Or-Parallelism. In *ICLP* Proceedings, pages 43–58, 2001.

[7] Y.S. Ramakrishnan, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, D.S. Warren: Efficient Model Checking using Tabled Resolution. In Proceedings of *Computer Aided Verification (CAV'97)*, pages 143–154 1997.

[8] H-F. Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan: Justification using Program Transformation. In Proceedings of *Logic Based Program Synthesis and Tranformation*, 2002.

[9] David S. Warren: Programming in Tabled Prolog (Draft Book). www.cs.sunysb.edu/~warren.

[10] Terrance Swift: Tabling for Non-Monotonic Programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4): 201-240, 1999.

[11] Günther Specht: Generating Explanation Trees even for Negations in Deductive Database Systems. Proc. of *the 5th Workshop of Logic Programming Environments*, 1992.

[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms. The MIT Press, 2001.

[13] A. Roychoudhury, C.R. Ramakrishnan, and I.V. Ramakrishnan: Justifying proofs using memo tables. *Second International ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP)*, pp. 178–189, 2000.

[14] Weidong Chen and David S. Warren: Query Evaluation under the Well Founded Semantics. *ACM Symposium on Principles of Database Systems*, pp. 168–179, 1993.

[15] Applied Logic Systems, Inc. http://www.als.com

A Tabling Engine Designed to Support Mixed-Strategy Evaluation

Ricardo Rocha Fernando Silva
DCC-FC & LIACC
Universidade do Porto, Portugal
{ricroc,fds}@ncc.up.pt

Vítor Santos Costa
COPPE Systems & LIACC
Universidade do Rio de Janeiro, Brasil
vitor@cos.ufrj.br

Abstract

Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing answers to subgoals. During tabled execution, there are several points where different operations can be applied. The decision on which operation to perform is determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The ability of using multiple strategies within the same evaluation can be a means of achieving the best possible performance. In this work, we present how the YapTab system was designed to support the two most successful tabling scheduling strategies: batched and local scheduling; and how it can be easily extended to support simultaneous mixed-strategy evaluation.

1 Introduction

The past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. One such proposal that has been gaining in popularity is the use of *tabling* or *tabulation* or *memoing*. Work on SLG resolution [2], as implemented in the XSB logic programming system [1], proved the viability of tabling technology for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, and Program Analysis. Tabling based models are able to reduce the search space, avoid looping, and have better termination properties than SLD based models.

The basic idea behind tabling is straightforward: programs are evaluated by storing answers of current subgoals in an appropriate data space, called the *table space*. The method then uses the table to verify whether calls to subgoals are repeated. Whenever such a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses.

During tabled execution, there are several points where we had to choose between continuing forward execution, backtracking, consuming answers from the table, or completing subgoals. The decision on which operation to perform is crucial to system performance and is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to different order of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *batched scheduling* and *local scheduling* [6].

Batched scheduling favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. On the other hand, local scheduling tries to complete subgoals sooner. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal in hand were resolved.

Empirical work from Freire *et al.* [6, 7] showed that, regarding the requirements of an application, the choice of the scheduling strategy can differently affect the memory usage, execution time and disk access patterns. Freire argues [5] that there is no single best scheduling strategy, and whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. As a means of achieving the best possible performance, Freire and Warren [8] proposed the ability of using multiple strategies within the same evaluation, by supporting mixed-strategy evaluation at the predicate level. However, to the best of our knowledge, no such implementation has yet been done.

In this work, we present how YapTab [10] was designed to support batched and local scheduling independently and how it can be easily extended to support simultaneous mixed-strategy evaluation. YapTab is a sequential tabling engine that extends Yap's execution model [12] to support tabled evaluation for definite programs. YapTab's implementation is largely based on the ground-breaking design of the XSB system [1], which implements the SLG-WAM [11].

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and discuss the differences between batched and local scheduling. We then present the support actually implemented in YapTab to deal with both scheduling strategies and discuss and it can be extended to support mixed-strategy evaluation.

2 Basic Tabling Definitions

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Whenever a tabled subgoal S is first called, an entry for S is allocated in the *table space*. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are found, they are stored into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to variant calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

Tabling based evaluation has four main types of operations for definite programs: entering a tabled subgoal; adding a new answer to a generator; exporting an answer from the table; and trying to complete a subgoal. In more detail:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table, and if not, adds a new entry for it and allocates a new generator node. Otherwise, it allocates a consumer node and starts consuming the available answers.

2. The *new answer* operation returns a new answer to a generator. It verifies whether a newly generated answer is already in the table, and if not, inserts it. Otherwise, it fails.

3. The *answer resolution* operation is executed every time the computation reaches a consumer node. It verifies whether newly found answers are available for the particular consumer node and, if any, consumes the next one. Answers are consumed in the same order they are inserted in the table. Otherwise, it *suspends* the current computation, either by freezing the whole stacks [11], or by copying the execution stacks to separate storage [4], and schedules a possible resolution to continue the execution.

4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. It executes when we backtrack to a generator node and all of its clauses have been tried. If the subgoal has been completely evaluated, the operation closes its table entry and reclaims space. Otherwise, it resumes one of the consumers with unconsumed answers.

Completion is needed in order to recover space and to support negation. We are most interested on space recovery in this work. Arguably, in this case we could delay completion until the very end of execution. Unfortunately, doing so would also mean that we could only recover space for consumers (suspended subgoals) at the very end of the execution. Instead we shall try to achieve *incremental completion* [3] to detect whether a generator node has been fully exploited, and if so to recover space for all its consumers.

Completion is hard because a number of generators may be mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*). Clearly, we can only complete SCCs together. We will usually represent an SCC through the oldest generator. More precisely, the youngest generator node which does not depend on older generators is called the *leader node*. A leader node is also the oldest node for its SCC, and defines the current completion point.

3 Scheduling Strategies

At several points we had to choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The actual sequence of operations thus depends on the scheduling strategy. We next discuss in some more detail batched and local scheduling.

3.1 Batched Scheduling

Batched scheduling takes its name because it tries to minimize the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues until it resolves all program clauses for the subgoal in hand. Only then the newly found answers will be returned to consumer nodes.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. Calls to non-tabled subgoals allocate interior nodes. First calls to tabled subgoals allocate generator nodes and variant calls allocate consumer nodes. However, if we call a variant tabled subgoal, and the correspondent subgoal is already completed, we can avoid consumer node allocation and instead perform what is called a *completed table optimization* [11]. This optimization allocates a node, similar to an interior node, that will consume the set of found answers executing compiled code directly from the table data structures associated with the completed subgoal [9].

When backtracking we may encounter three situations: **(i)** if backtracking to a generator or interior node, we take the next available alternative; **(ii)** if backtracking to a consumer node, we take the next unconsumed answer; **(iii)** if there are no available alternatives or no unconsumed answers, we simply backtrack to the previous node on the current branch. Note however that, if the node without alternatives is a leader generator node, then we must check for completion.

In order to perform completion, we must ensure that all answers have been returned to all consumers in the SCC. The process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

At engine level, the *fixpoint check procedure* is controlled by the leader of the SCC. The procedure traverses the consumer nodes in the SCC in a bottom-up manner to determine whether the SCC has been completely evaluated or whether further answers need to be consumed. Initially, it searches for the bottom consumer node with unresolved answers, and as long as there are available answers, it will consume them. After consuming the available set of answers, the consumer suspends and fails into the next consumer with unresolved answers. This process repeats until it reaches the last consumer node, in which case it fails into the leader node in order to allow the re-execution of the fixpoint check procedure. When a fixpoint is reached, all subgoals in the SCC are marked completed and the stack segments belonging to the completed subtree are released.

3.2 Local Scheduling

Local scheduling is an alternative tabling scheduling strategy that tries to complete subgoals sooner. Evaluation is done one SCC at a time, and answers are returned outside of a SCC only after that SCC is completely evaluated. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal in hand were resolved. We next present in Fig. 1 a small example that clarifies the differences between batched and local evaluation.

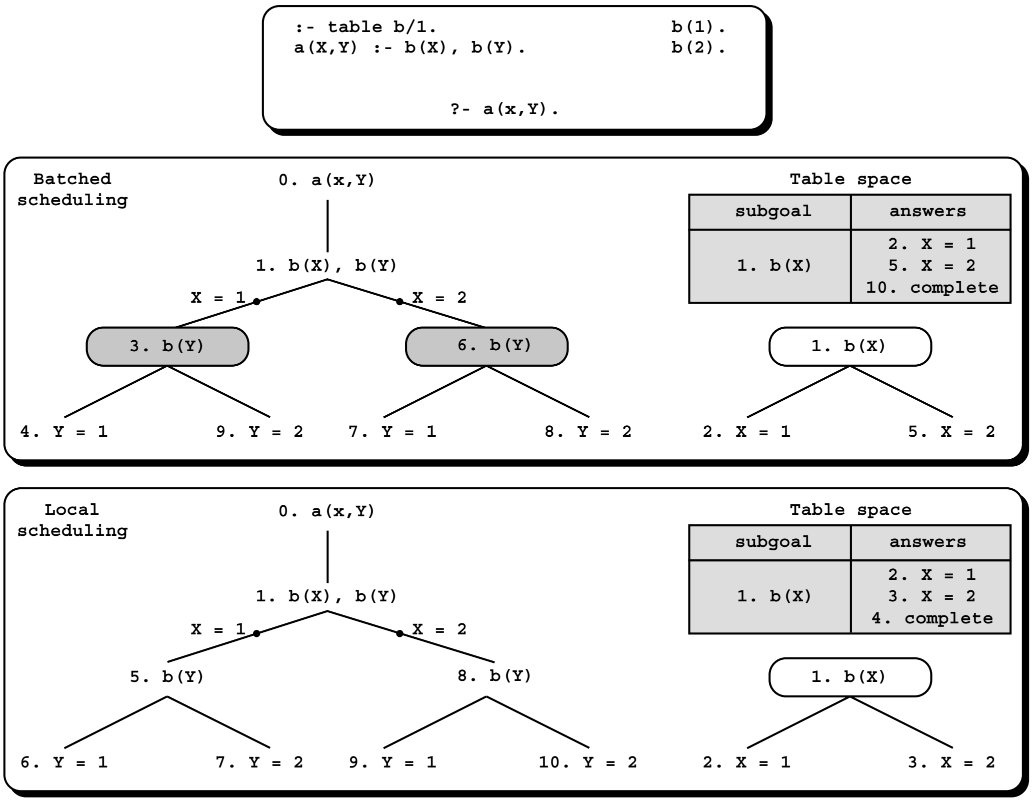


Figure 1: Batched versus local scheduling

At the top, the figure illustrates the program code and the query goal used in both evaluations. Declaration `:- table b/1` in the program code indicates that calls to predicate `b/1` should be tabled. The two sub-figures below depict the evaluation sequence for each scheduling strategy, which includes the resulting table space and forest of trees. The numbering of nodes denotes the evaluation sequence. The leftmost tree represents the original invocation of the query goal `a(X,Y)`. As we shall see, computing `a(X,Y)` requires computing `b(X)`. For simplicity of presentation, the computation tree for `b(X)` is represented independently at the right. We next describe in more detail the two evaluations.

In both cases, the evaluation begins by resolving the query goal against the unique clause for predicate `a/2`, thus calling the tabled subgoal `b(X)`. As this is the first call to `b(X)`, we create a generator node (generators are depicted by white oval boxes) and insert a new entry in the table space for it. The first clause for `b(X)` immediately succeeds, obtaining a first answer for `b(X)` that is stored in the table (step 2). The interesting aspect that results from the figure, is how both strategies handle the continuation of the evaluation of `b(X)`.

For batched scheduling, the evaluation proceeds executing as in standard Prolog with the continuation call `b(Y)`, therefore creating consumer node 3 (consumers are depicted by gray oval boxes). Node 3 is a variant call to `b(X)`, so instead of resolving the call against the program clauses, we consume answers from the table space. As we already have one answer stored in the table for this call (`X=1`), we continue by consuming the available answer, which leads to a first solution for the query goal (`X=1;Y=1`). When returning to node 3, we must suspend the consumer node because there are no more answers for it in the table. We then backtrack to node 1 to try the second clause for `b(X)`, and a new answer is found (`X=2`). In the continuation, a new consumer is created (node 6) and two new solutions are found for the query goal (steps 7 and 8). Node 6 is then suspended and the computation backtracks again to node 1. At that point, we can check for completion. However, the generator cannot complete because consumer 3 has unconsumed answers. The computation is then resumed at node 3 and a new solution for the query goal is found (step 9). When returning to the generator node 1, we can finally complete the tabled subgoal call `b(X)` (step 10).

On the other hand, for local scheduling, the evaluation fails back after the first answer was found (step 2) in order to find the complete set of answers for `b(X)` and therefore complete before returning answers to the calling environment. We thus backtrack to node 1, execute the second clause for `b(X)`, and find a second answer for it (step 3). Then, we fail again to node 1, and the tabled subgoal call `b(X)` can be completed (step 4). The two found answers are consumed next by executing compiled code directly from the table structure associated with the completed subgoal `b(X)`. The variant calls to `b(X)` at steps 5 and 8 are also resolved by executing compiled code from the table.

In batched scheduling, when a new answer is found, variable bindings are automatically propagated to the calling environment. For some situations, this behavior may result in creating complex dependencies between consumers. On the other hand, the clear advantage of local scheduling shown in the example does not always hold. Since local scheduling delays answers, it does not benefit from variable propagation, and instead, when explicitly returning the delayed answers, it incurs an extra overhead for copying them out of the table. Local scheduling does perform arbitrarily better than batched scheduling for applications that benefit from answer subsumption, that is, where we delete non-minimal answers every time a new answer is added to the table. On the other hand, Freire *et al.* [6] showed that, on average, local scheduling is about 15% slower than batched scheduling in the SLG-WAM. Similar results were also obtained for batched and local scheduling in YapTab [10].

4 Implementation

We next give a brief introduction to the implementation of YapTab. Throughout, we focus on the support for the two tabling scheduling strategies.

The YapTab design is very close to the original SLG-WAM [11]: it introduces a new data area, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations: *tabled subgoal call*, *new answer*, *answer resolution*, and *completion*. The substantial differences between the two designs reside in the data structures and algorithms used to control the process of leader detection and scheduling of unconsumed answers. The SLG-WAM considers that such control should be done at the level of the data structures corresponding to first calls to tabled subgoals, and it does so by associating *completion frames* to generator nodes. It uses a *completion stack* of generators to detect completion points. Essentially, the *completion stack* stores information about the generator nodes and the dependencies between them. Each time a new generator is introduced it becomes the current leader node. Each time a new consumer is introduced one verifies if it is for an older generator node $\mathcal{G}$. If so, $\mathcal{G}$'s leader node becomes the current leader node.

On the other hand, YapTab innovates by considering that the control of leader detection and scheduling of unconsumed answers should be performed through the data structures corresponding to variant calls to tabled subgoals, and it associates a new data structure, the *dependency frame*, to consumer nodes. We believe that managing dependencies at the level of the consumer nodes is a more intuitive approach that we can take advantage of. The new data structure allows us to eliminate the need for a separate completion stack and to slightly improve the fixpoint check procedure. In the SLG-WAM, each step of the fixpoint check procedure is done by traversing the consumers in a SCC by groups, with each group corresponding to consumers for a common variant subgoal. YapTab simplifies by considering the whole set of consumers within a SCC as a single group, and it thus traverses the whole set in a single pass.

4.1 Table Space

The table space can be accessed in different ways: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is in the table, and if not insert it; to forward answers to consumer nodes; and to mark subgoals as completed. Hence, a correct design of the algorithms to access and manipulate the table is a critical issue to obtain an efficient implementation. Our implementation uses tries as the basis for tables, as proposed by Ramakrishnan *et al.* [9]. Tries provides complete discrimination for terms and permits lookup and possibly insertion to be performed in a single pass through a term.

Figure 2 shows the general table structure for a tabled predicate. Table lookup starts from the *table entry* data structure. Each table predicate has one such structure, which is allocated at compilation time. A pointer to the table entry can thus be included in the compiled code. Calls to the predicate will always access the table starting from this point.

The table entry points to a tree of trie nodes, the *subgoal trie structure*. More precisely, each different call to the tabled predicate in hand corresponds to a unique path through the subgoal trie structure. Such a path always starts from the table entry, follows a sequence of subgoal trie data units, the *subgoal trie nodes*, and terminates at a leaf data structure, the *subgoal frame*.

Each subgoal frame stores information about the subgoal, namely an entry point to its *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different answer to the entry subgoal. All answer leave nodes are chained together in insertion time order in a linked list, so that we can recover answers in the same order they were

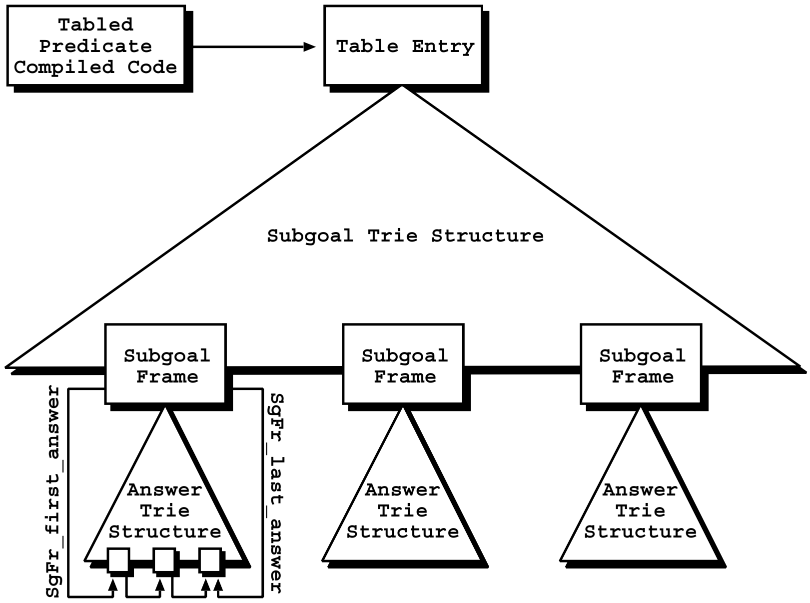


Figure 2: Using tries to organize the table space

inserted. The subgoal frame points at the first and last entry in this list. A consumer node thus needs only to point at the leaf node for its last consumed answer, and consumes more answers just by following the chain of leaves.

4.2 Tabled Nodes

In YapTab, applying batched or local scheduling to a tabled evaluation only depends on the way generators are implemented. All the other tabling extensions are commonly used for both strategies without any modifications. As we shall see, this makes YapTab highly suitable to support mixed-strategy evaluation.

Remember that interior nodes are implemented as WAM choice points [13]: the `CP_TR`, `CP_H`, `CP_B`, `CP_CP`, `CP_AP` and `CP_ENV` choice point fields are used to store at choice point creation, respectively, the top of trail; top of global stack; failure continuation choice point; success continuation program counter; choice point next alternative; and current environment. Generator and consumer nodes are implemented as WAM choice points extended with some extra fields to control tabling execution.

To implement consumer nodes we extended the WAM choice points with the dependency frame data structure. Dependency frames store the last consumed answer for the correspondent consumer node; and information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers.

To prevent answers from being returned to the calling environment of a generator node, after a new answer is found for a particular tabled subgoal, local scheduling fails and backtracks in order to search for the complete set of answers. These answers are consumed later when all program clauses for the subgoal in hand were resolved. Therefore, when backtracking to a generator node without alternatives, we must also act like a consumer node to consume the set of found answers. Thus, for local scheduling, generator choice points are also extended with dependency frames. For batched scheduling we only need to access the subgoal frame where answers should be stored, so generators are implemented as WAM choice points extended with a pointer to the corresponding subgoal frame, the `CP_SgFr` field. Figure 3 illustrates how consumers and generators are differently handled to support batched and local scheduling.

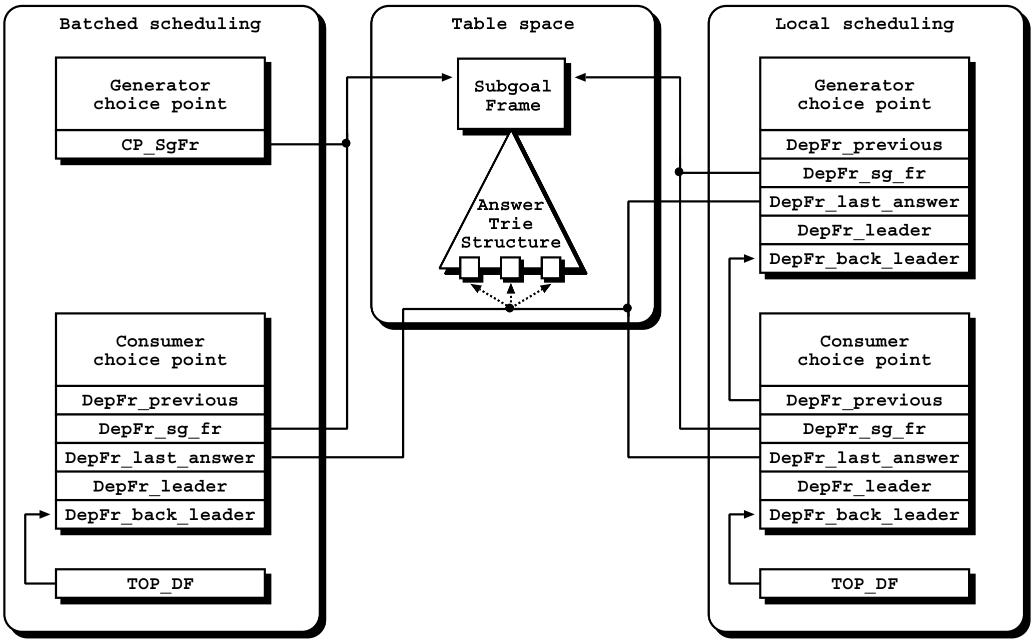


Figure 3: Consumers and generators with batched and local scheduling

Each dependency frame is a five field data structure. The `DepFr_previous` is a pointer to the previous dependency frame on stack and it allows to form a list of dependency frames on stack. A global `TOP_DF` variable points to the youngest dependency frame on stack. The `DepFr_sg_fr` and the `DepFr_last_answer` are pointers respectively to the correspondent subgoal frame and to the last consumed answer, and they are used to connect choice points with the table space in order to search for and to pick up new answers. Moreover, for local scheduling, we use the `DepFr_sg_fr` field of the dependency frame to access the correspondent subgoal frame. For batched scheduling, we use the new `CP_SgFr` choice point field. The `DepFr_leader` and the `DepFr_back_leader` are pointers respectively to the leader node at creation time and to the leader node where we perform the most recent unsuccessful completion operation, and they are used to support the fixpoint check procedure. The dependency frame `DepFr_leader` field is initialized by the `compute_leader()` procedure, whilst the `DepFr_back_leader` field is initialized with a `NULL` value. Their use is detailed next.

4.3 Answer Resolution

The answer resolution operation should be executed every time the computation fails back to a consumer. To achieve this, when a new consumer choice point is allocated, its `CP_AP` field is made to point to the `answer_resolution` instruction. Figure 4 shows the pseudo-code for it.

Initially, the procedure checks the table space for unconsumed answers. If there are new answers, it loads the next available answer and proceeds. Otherwise, it schedules for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. This is the case when the `DepFr_back_leader` field is `NULL`. Otherwise, we know that the computation has been resumed from an older leader node $\mathcal{L}$ during an unsuccessful

```
answer_resolution (consumer node CN) {
  DF = dependency_frame_for(CN)
  if (DepFr_last_answer(DF) != SgFr_last_answer(DepFr_sg_fr(DF)))
    load_next_unconsumed_answer_and_proceed()
  back_cp = DepFr_back_leader(DF)
  if (back_cp == NULL)
    backtrack()
  df = DepFr_previous(DF)
  while (consumer_for(df) is younger than back_cp) {
    if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
      // move to previous consumer with unconsumed answers
      DepFr_back_leader(df) = back_cp
      move_to(consumer_for(df))
    }
    df = DepFr_previous(df)
  }
  // move to older leader node
  move_to(back_cp)
}
```

Figure 4: Pseudo-code for `answer_resolution()`

completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and that is younger than $\mathcal{L}$. We do this by restoring bindings and stack pointers. If no such consumer node can be found, backtracking must be done to node $\mathcal{L}$.

The process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

4.4 Leader Nodes

The completion operation takes place when we backtrack to a generator node that **(i)** has exhausted all its alternatives and that **(ii)** is a leader node (remember that the youngest generator which does not depend on older generators is called a leader node). We designed novel algorithms to quickly determine whether a generator node is a leader node. The key idea in our algorithms is that each dependency frame holds a pointer to the resulting leader node of the SCC that includes the correspondent consumer node. Using the leader node pointer from the dependency frames, a generator can quickly determine whether it is a leader node. More precisely, a generator $\mathcal{L}$ is a leader node when either **(a)** $\mathcal{L}$ is the youngest tabled node, or **(b)** the youngest consumer says that $\mathcal{L}$ is the leader.

Our algorithm thus requires computing leader node information whenever creating a new consumer node $\mathcal{C}$. We proceed as follows. First, we hypothesize that the leader node is $\mathcal{C}$'s generator, say $\mathcal{G}$. Next, for all consumer nodes older than $\mathcal{C}$ and younger than $\mathcal{G}$, we check whether they depend on an older generator node. Consider that there is at least one such node and that the oldest of these nodes is $\mathcal{G}'$. If so then $\mathcal{G}'$ is the leader node. Otherwise, our hypothesis was correct and the leader node is indeed $\mathcal{G}$. Leader node information is implemented as a pointer to the choice point of the newly computed leader node. Figure 5 shows the procedure that computes the leader node information for a new consumer.

The procedure traverses the dependency frames for the consumer nodes between the new consumer and its generator in order to check for older dependencies. As an optimization it only searches until it finds the first dependency frame holding an older reference (the `DepFr_leader` field). The nature of the procedure ensures that the remaining dependency frames cannot hold older references.

For local scheduling, when we store a new generator node $\mathcal{G}$ we also allocate a dependency frame. As an optimization we can avoid calling `compute_leader()` to initialize the `DepFr_leader`

```
compute_leader (consumer node CN) {
  DF = dependency_frame_for(CN)
  leader_cp = generator_for(CN)
  df = TOP_DF
  while (consumer_for(df) is younger than leader_cp ) {
    if (leader_cp is equal or younger than DepFr_leader(df)) {
      // found older dependency
      leader_cp = DepFr_leader(df)
      break
    }
    df = DepFr_previous(df)
  }
  DepFr_leader(DF) = leader_cp
}
```

Figure 5: Pseudo-code for `compute_leader()`

field, because it will always compute $\mathcal{G}$ as the leader node.

4.5 Completion with Batched Scheduling

When a generator choice point tries the last program clause, its `CP_AP` field is updated to the `completion` instruction. Since then, every time we backtrack to the choice point the instruction gets executed. Figure 6 shows the pseudo-code that implements completion for batched scheduling.

```
completion (generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (consumer_for(df) is younger than GN)) {
      if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        // move to first consumer with unconsumed answers
        DepFr_back_leader(df) = GN
        move_to(consumer_for(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
  }
  backtrack()
}
```

Figure 6: Pseudo-code for `completion()` with batched scheduling

Initially, the procedure finds out if the generator is the current leader node. If not, it simply backtracks to the previous node. Being leader, it checks whether all younger consumer nodes have consumed all their answers. To do so, it walks the chain of dependency frames looking for a frame which has not yet consumed all the generated answers. If there is such a frame, the computation should be resume to the corresponding consumer node. Otherwise, it can perform completion. This includes **(i)** marking as complete all the subgoals in the SCC; **(ii)** deallocating all younger dependency frames; and **(iii)** backtracking to the previous node to continue the execution.

4.6 Completion with Local Scheduling

To implement completion for local scheduling, we only need to slightly change the previous procedure. Figure 7 shows the modified pseudo-code.

```
completion (generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (consumer_for(df) is younger than GN) {
      if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        // move to first consumer with unconsumed answers
        DepFr_back_leader(df) = GN
        move_to(consumer_for(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
    completed_table_optimization()                              // new
  }
  CP_AP(GN) = answer_resolution                                 // new
  load_first_unconsumed_answer_and_proceed()                    // new
}
```

Figure 7: Pseudo-code for `completion()` with local scheduling

There is a major change to the completion algorithm for local scheduling. As newly found answers cannot be immediately returned, we need to consume them at a later point. If we perform completion successfully, we start consuming the set of answers that have been found by executing compiled code directly from the trie data structure associated with the completed subgoal. Otherwise, we must act like a consumer node and start consuming answers.

5 Discussion

In result of its clear design based on the dependency frame data structure, YapTab already includes all the machinery required to support batched and local scheduling simultaneously. Extending YapTab to use multiple strategies at the predicate level is straightforward. Only two new features have to be addressed: **(i)** support strategy-specific Prolog declarations like '`:- batched path/2.`' in order to allow the user to define the strategy to be used to resolve the subgoals of a given predicate; **(ii)** at compile time generate appropriate tabling instructions, such as `batched_new_answer` or `local_completion`, accordingly to the declared strategy for the predicate. With these two simple compiler extensions we are able to use all the algorithms described and already implemented for batched and for local scheduling without any further modification.

The proposed data structures and algorithms can also be easily extended to support different strategies per predicate, that is, allow the user to define the strategy to be used to resolve each subgoal. Moreover, they can be extended to support dynamic switching from batched to local scheduling, while a generator is still producing new answers. However, further work is still needed to study if there is a use for such flexibility.

In this work we concentrated on the issues concerning the design and implementation of both strategies. Currently, we have already batched and local scheduling functioning separately in YapTab and we are now working on adjusting the system for mixed-strategy evaluation. After having the system implementing mixed-strategy evaluation we plan to use a set of common tabled benchmarks to investigate and study the impact of combining both strategies for tabled evaluation.

Acknowledgments

This work has been partially supported by APRIL (POSI/SRI/40749/2001), $CLoP^n$ (CNPq), PLAG (FAPERJ), and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References

[1] The XSB Logic Programming System. Available from `http://xsb.sourceforge.net`.

[2] W. Chen, M. Kifer, and D. S. Warren. Hilog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[3] W. Chen, T. Swift, and D. S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[4] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, 2000.

[5] J. Freire. *Scheduling Strategies for Evaluation of Recursive Queries over Memory and Disk-Resident Data*. PhD thesis, Department of Computer Science, State University of New York, 1997.

[6] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 243–258. Springer-Verlag, 1996.

[7] J. Freire, T. Swift, and D. S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*, pages 198–212, 1997.

[8] J. Freire and D. S. Warren. Combining Scheduling Strategies in Tabled Evaluation. In *Workshop on Parallelism and Implementation Technology for Logic Programming*, 1997.

[9] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[10] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[11] 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):586–634, 1998.

[12] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 261–267. Springer-Verlag, 1999.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

Alternatives for compile & run in the WAM

Remko Tronçon Gerda Janssens Bart Demoen
K.U.Leuven – Department of Computer Science
{remko,gerda,bmd}@cs.kuleuven.ac.be

Abstract

Decision tree learning algorithms dynamically generate huge queries. Because these queries are executed often, the trade-off between meta-calling and compiling & running them has been in favor of the latter, as compiled code is faster. However, compilation is expensive, and experiments show that sometimes meta-call can outperform compile & run. In this paper, we investigate alternative approaches that either improve meta-call execution, or reduce compilation time without sacrificing execution speed. By embedding the meta-call we can improve its execution by a factor of 3 to 4. We also propose a hybrid scheme of compilation and meta-call that reduces compilation times by an order of magnitude. Our results strongly suggest that the same techniques are worth applying in the context of decision tree learners.

1 Introduction

In the context of inductive learning, we are involved in a system named ilProlog [1], which is a Prolog system with a WAM [8] based abstract machine emulator (written in C), and with special support for Inductive Logic Programming (ILP). One of its features is the use of *query packs* [3]. Such a query pack is basically the body of a rule with no arguments, containing a huge number of disjunctions. The query pack execution in ilProlog deals with the disjunctions in a special way, e.g. by avoiding a branch when it has already succeeded before. These query packs are generated dynamically by the ILP system, compiled by the underlying Prolog system, and the compiled code is executed on a dataset (which is actually a large collection of different logic programs). This is not an unreasonable approach, and we have indeed measured large speedups in ILP systems based on this approach [3].

The compilation of a query pack takes in many cases more time than its execution, and since query pack generation happens incrementally, we have investigated incremental compilation of query packs. However, incremental query pack compilation requires a tight coupling between the generation of a query pack and its compilation. On top of that, matters are complicated by the fact that in between the generation phase and the compilation phase, a query pack transformation pass can be helpful for performance reasons [4]. Maintaining the coupling between generation and compilation in the presence of the transformations became virtually impossible, and the alternative seemed to fall back on meta-interpretation of the generated queries as in early versions of the system. But compilation and subsequent execution results in a speedup: folklore says indeed that meta-interpretation is ten (or more) times slower than normal execution. So we were faced with a choice: give up speed for added flexibility (i.e. allow query pack transformations and interpret the resulting queries) or gain speed (by compilation) but lose the query pack transformations. We followed another alternative and attempted to speed up meta-interpretation to the point that (full) compilation is no longer necessary. The result of this research is two-fold: in Section 4 we present an extremely fast meta-interpreter that can be described in three lines of (new) abstract machine code; in Section 5 we present a hybrid scheme in which a query pack is compiled *partially*: only the control flow is compiled. The compilation times are much smaller than the full fledged compilation described in [7], and the execution has the same efficiency as fully compiled code, or better. We start by showing in Section 3 that, contrary to the common belief, meta-interpretation of a program can be more efficient than the execution of compiled code for the same program.

Although our motivation lies in optimizing ILP execution systems using features such as query packs, we will for now focus on queries containing just regular conjunctions and disjunctions. Future work will extend the ideas to special disjunctions used in query packs. This means that this paper is of general interest to any plain WAM implementation.

We assume knowledge of Prolog and its implementation. For a good introduction to the WAM, see [2].

2 Background & Motivation

We will start by sketching a particular setting in which this work is relevant, namely the execution of queries in Inductive Logic Programming.

The goal of Inductive Logic Programming is to find patterns in a large set of data (or examples). In ILP, each example is a logic program, and the patterns are represented as logic queries. The way in which ILP searches for these patterns is basically generate-and-test: generated queries are run on all examples; based on the failure or success of these queries, only the ones with the 'best' results[1] are kept and are extended (e.g. by adding a literal in the back). These newly generated queries are in turn tested on each example, and this process continues until a satisfactory query (or a set of queries) describing the examples has been found.

At each step, a set of queries is executed on a large set of logic programs (the examples). Moreover, since these queries are the result of adding different literals to the end of another query, the queries in this set have a lot of common prefixes. To avoid repeating the common parts by executing each query separately, the set of queries is transformed into a special kind of disjunction, a query pack [3]. For example, the set of queries

```
?- a, b, c, d.
?- a, b, c, e.
?- a, b, f, g.
```

will be transformed into the query

```
?- a, b, ( (c,(d;e)) ; f ).
```

However, because only the success of a query on an example is measured, the normal Prolog disjunction might still cause too much backtracking, so the ';'/2 has slightly different semantics in query packs, and cuts away queries from the disjunction as soon as they succeed.

Since each query pack is going to be run on a large set of examples, it seems straightforward to compile the query pack first, and run the compiled code on the examples. However, as mentioned in the introduction, compilation takes more time than execution, and on top of that hinders exploiting the incremental nature of query packs. Therefore, we will investigate alternatives for this compile & run approach.

[1] which queries are best depends on the ILP algorithm.

	SICStus Prolog	Yap	ilProlog	hProlog
compiled	15.00	22.05	19.22	13.64
call	4.91	3.91	1.68	1.59
conj_call	8.95	2.63	1.14	1.00

Table 1: Normalized execution times for running a query in 3 different ways

3 Meta-call

A first alternative to executing compiled code, is to simply meta-call the query. While this might seem a slow alternative, we will start by showing that this approach is not always unreasonable.

Consider the following program:

```
a :- b, f(<big_term>).
b.
...
b.
f(_) :- fail.
```

where predicate b has a number of alternatives, say N, and <big_term> is a large term. More specific, let N be 20, and <big_term> a list of 60 different terms of the form $g(0)$.

Suppose that we want to activate the body of a. A normal way of doing this is to query a:

```
?- a.
```

This will execute the compiled body of a. However, there are two alternatives for this. Firstly, we can meta-call the goal which consists of the body of a as a Prolog term:

```
?- call((b, f(<big_term>))).
```

Secondly, we can also use the information we have about generated queries: they are always conjunctions which do have a left side which is a simple (callable) goal. Instead of calling the body of a using `call`, we can use a special predicate *conj_call*, which is defined as

```
conj_call((X,Y)) :- !, call(X), conj_call(Y).
conj_call(X) :- call(X).
```

Trying these three approaches in a number of different Prolog systems and measuring the execution time (without compilation) results in the timings shown in Table 1. The timings are normalized with respect to the execution of *conj_call* in hProlog. ilProlog and hProlog are two descendants of dProlog [5]; the reason for including hProlog in the measurements is that hProlog has an ISO-compatible meta-call like SICStus Prolog, while ilProlog and Yap do not type-check all arguments of a conjunction to be callable goals before executing the conjunction (as the ISO Prolog standard imposes).

In all cases, we see that the meta-call is faster than the compiled call, and that the specialized *conj_call* is even faster[2]. The explanation is simple: the construction of the <big_term> is performed over and over again in the compiled version, and not even once in the meta-called versions.

[2]except for SICStus Prolog: the reason is extra overhead due to the module system and term_expansion.

Since we have control over the generation of the body (it is one query from a query pack), we can try variations like representing a conjunction as a list of its conjuncts and use a list version of *conj_call*:

```
conj_call([]).
conj_call([X|Y]) :- call(X), conj_call(Y).
```

This gives extra speedup in all systems, because list traversal is implemented better than conjunction traversal.

One could think that the situation can be easily improved for the compiled version, by compiling it as if its code where:

```
a :- X = <big_term>, b, f(X).
```

However, such a transformation - although correct - can make performance worse: if *b* fails, the term has been constructed in vain. Moreover, the memory requirement can become arbitrarily larger if this transformation is performed systematically in a Prolog program.

From these results, we can conclude that compile&run is not a priori the fastest alternative, even without taking into account the overhead of compilation. However, the normal meta-call and the *conj_call* do not always perform as well as the execution of compiled code, so we will try optimizing this in the following section. Table 1 also shows that ilProlog has already excellent meta-call performance, so any improvement on it is relevant.

4 Embedding the meta-call

To get more speedup from a specialized meta-call such as *conj_call*, it should be implemented in the internals of the Prolog system. One could choose to implement the specialized call completely in the host language of the system, but it is conceptually clearer to implement a series of new emulator instructions and to use those to implement *conj_call*. Also, this approach results in the same performance.

Taking into account the knowledge we have about the input to the first version of *conj_call*, its WAM-code is:

```
    gettbreg A2                        0
    switchonterm struct=L1, else=L2    *
L1: get_structure A1 ,/2               *
    unitvar A1                         *
    allocate 3                         *
    unipvar Y2                         *
    puttbreg A2                        0
    call call/1 (user)                 *
    putpval Y2 A1                      #
    deallex conj_call/1                #
L2: execute call/1                     *
```

We can make a new instruction `mc_switch`, which performs all the actions from the instructions labeled with *, and another new instruction `mc_continueconj` that performs the actions of the instructions labeled with #. The instructions labeled with 0 have no more function, and are simply deleted. This leads to the code:

```
L1: mc_switch L2
L2: mc_continueconj L1
```

The instruction mc_switch distinguishes 2 types of terms in the first argument register: a ','/2 term and a goal. In the latter case, the goal is simply called. When the argument is a conjunction, an environment is allocated, and the first argument of ','/2 is called with the label of mc_continueconj (passed through the argument of mc_switch) as its continuation. When execution reaches mc_continueconj, the environment created by mc_switch is deallocated, the continuation pointer is restored, and mc_switch is executed with the second argument of ','/2.

Let's now extend the *conj_call* from the previous section to an embedded meta-call which can handle disjunctions. The Prolog-code for such a predicate would be:

```
conjdisj_call((X,Y)) :- !, call(X), conjdisj_call(Y).
conjdisj_call((X;Y)) :- !, ( conjdisj_call(X) ; conjdisj_call(Y) ).
conjdisj_call(X)     :- call(X).
```

Note that since the disjunctions might in turn contain conjunctions in both arguments, we cannot assume that the first argument of ';'/2 is a simple goal as we could with ','/2.

To implement this in the WAM, the mc_switch instruction has to be extended, and an extra instruction mc_continuedisj is introduced. The resulting code is as follows.

```
L1: mc_switch L2 L3
L2: mc_continueconj L1
L3: mc_continuedisj L1
```

The only extension mc_switch needs is the ability to handle a third type of term in the first argument register: a ';'/2 term. In this case, a choice-point is created with the label of mc_continuedisj (passed through the second argument of mc_switch) as an alternative, after which mc_switch is executed on the first argument of ';'/2. In mc_continjuedisj, the choice-point is removed, and mc_switch is executed with the second argument of ';'/2.

For subsequent goals in a conjunction, the embedded meta-call will do a deallocate of an environment (in mc_continueconj), immediately followed by an allocate (in mc_switch). Such redundancy can be avoided by adding an extra test: mc_continueconj checks the next functor, and skips the deallocate and allocate if it is again a conjunction.

As results will show, this new 'embedded' meta-call results in a speedup of factor 3 to 4 over normal meta-call. However, as will be clear from the performance evaluation in Section 6, it is not fast enough. Also, the implementation of an embedded meta-call suffers from the drawback that extending it to better deal with built-ins requires adding more checks in the mc_switch instruction, which is both cumbersome and likely to slow down the execution.

5 Control flow compilation

The major reason why meta-call can be competitive with running a compiled query is that the code for the compiled query contains instructions for setting up the arguments of the called goals – the `put` instructions. This requires costly emulator cycles in compiled code, while setting up the arguments for the meta-called goal happens in the same emulator cycle as the call itself. Moreover, compilation itself is costly due to the non-linear allocation tasks such as assigning variables to environment slots, managing argument registers, ...

It would be interesting to combine this advantage of meta-interpretation (avoiding to set up arguments to goals using `put` instructions), with a simple form of compilation without expensive operations such as register allocation. Such a simple compiler would amongst others have the advantage that it can inline built-ins, and would be easy to extend. For this purpose, we introduce *control flow compilation*. The idea is to generate code for a query which describes the flow of control, but where the goals themselves are still meta-called in the sense that their arguments have been preconstructed on the heap before the execution has started. The code generated by control flow compilation will look very much like ordinary code, but it will not contain any instructions related to arguments of goals, neither to variables. We illustrate the idea by a sequence of steps that will lead to the desired compilation.

Given the query *?- a(X,Y),(b(Y,Z);c(Y,Z);d(Y,Z)).* We can flatten this out into the structure *query(a(X,Y),b(Y,Z),c(Y,Z),d(Y,Z))*, and generate a predicate `cf_call` which, given this term as an argument, executes the original query:

```
cf_call(Query) :-
    arg(Query,1,G1),
    call(G1),
    ( arg(Query,2,G2),        ?- cf_call(query(a(X,Y),b(Y,Z),c(Y,Z),d(Y,Z))).
      call(G2)
    ; arg(Query,3,G3),
      call(G3)
    ; arg(Query,4,G4),
      call(G4)
    ).
```

Notice how this code reflects the structure of the query, but calls the individual goals using meta-call. Compiling this predicate results in the following WAM code:

```
     allocate 3
     getpvar Y2 A1
     put_int A2 1               *
     putpval Y2 A3              *
     builtin_arg_3 A3 A2 A1     *
     call call/1                *
     trymeorelse L1
     put_int A2 2               #
     putpval Y2 A3              #
     builtin_arg_3 A3 A2 A1     #
     deallex call/1             #
L1:  retrymeorelse L2
```

```
       put_int A2 3                #
       putpval Y2 A3               #
       builtin_arg_3 A3 A2 A1      #
       deallex call/1              #
L2:    trustmeorelsefail
       put_int A2 4                #
       putpval Y2 A3               #
       builtin_arg_3 A3 A2 A1      #
       deallex call/1              #
```

This block of instructions starts by allocating the environment on the stack, and putting the argument of the predicate (the query/4 term) into the second variable slot of the environment, Y2. Then, each group of instructions labeled with * and # represent the call to arg/3, immediately followed by a call to call/1. Except for the varying integer argument for arg/3 and the `call` or `deallex` instruction in the end, these groups are identical. We therefore merge the instructions labeled * into a new instruction `arg_call`, and the instructions labeled # into `arg_deallex`. Both instructions fetch a given argument from the structure in Y2, put it in the first argument register for call/1, and finally either call call/1 (for `arg_call`) or deallocate the environment and execute call/1 (for `arg_deallex`). If we use these new instructions, the new code for *cf_call* becomes:

```
       allocate 3
       getpvar Y2 A1
       arg_call 1
       trymeorelse L1
       arg_deallex 2
L1:    retrymeorelse L2
       arg_deallex 3
L2:    trustmeorelsefail
       arg_deallex 4
```

What remains is essentially only the control flow of the original query as can be seen in Figure 1. Generating both the WAM code for *cf_call* and the flat query structure used in this code is simple, and can be done in linear time without the need of the full fledged compiler.

The standard control flow compilation scheme leaves room for some optimizations. The implementation `arg_call` and `arg_deallex` uses call/1 which, for a goal of arity n, loops over all n arguments of the goal, putting them in argument registers, and finally performs the call to the goal. Since it is reasonable to assume that the goals will have a limited amount of arguments most of the time, we can specialize `arg_call` for goals with arguments under this limit by unrolling the argument loop, and as such avoid a loop during each meta-call.

6 Performance measurements

The experiments we describe in this section were performed with ilProlog. The ilProlog system was extended to support the two new approaches proposed in this paper: new abstract machine instructions were implemented, and a light-weight compiler for the control flow compilation was added. For both the compile & run and the control flow compilation approach, we used the facility of ilProlog to load compiled code directly into the code area without writing it to a file. The

	Compiled code	Control flow code
	⋮	⋮
	`bldtvar A1`	
	`putpvar Y2 A2`	*arg_call 1*
	`call a/2`	
	`trymeorelse L1`	`trymeorelse L1`
	`putpval Y2 A1`	
	`bldtvar A2`	*arg_deallex 2*
	`deallex b/2`	
`L1:`	`retrymeorelse L2`	`retrymeorelse L2`
	`putpval Y2 A1`	
	`bldtvar A2`	*arg_deallex 3*
	`deallex c/2`	
`L2:`	`trustmeorelsefail`	`trustmeorelsefail`
	`putpval Y2 A1`	
	`bldtvar A2`	*arg_deallex 4*
	`deallex d/2`	

Figure 1: Compiled code & 'control flow code' for *a(X,Y),(b(Y,Z);c(Y,Z);d(Y,Z))*

experiments were run on a Pentium III 660 Mhz with 256 Mb main memory running Linux under a normal load.

We generated benchmark conjunctions and disjunctions. We used two kinds of calls: either all calls had ground arguments (e.g. *a(1,1),a(1,1))*, or the calls had variables as arguments and consecutive calls shared variables (e.g. *a(X,Y),a(Y,Z)* or *a(X,Y), a(Y,Z), (a(Z,A),a(A,B) ; a(Z,C),a(C,D) ; a(Z,E), a(E,F)))*. In order to judge the performance of our approaches, the execution time spent in the called predicates was made minimal by using deterministic predicate definitions that are actually just facts (e.g. the fact a(_X,_Y). is compiled into a `proceed`).

The tables report on the following benchmarks:

- conj_sh_65: a conjunction of 65 calls with shared variables.
- conj_gr_65: a conjunction of 65 ground calls.
- conj_sh_130: a conjunction of 130 calls with shared variables.
- conj_gr_130: a conjunction of 130 ground calls.
- disj_sh_5: a disjunction with branching factor 3, 5 consecutive calls in a branch, with shared variables between consecutive calls and a total of 65 calls in the disjunction.
- disj_gr_5: same as disj_sh_5, but with ground calls.
- disj_sh_10: a disjunction with branching factor 3, 10 consecutive calls in a branch, with shared variables between consecutive calls and a total of 130 calls in the disjunction.
- disj_gr_5: same as disj_sh_10, but with ground calls.

	conj				disj			
	sh_65	gr_65	sh_130	gr_130	sh_5	gr_5	sh_10	gr_10
comp	1.155	1.366	2.165	2.576	1.069	1.178	1.709	1.987
call	9.485	9.508	18.968	18.906	7.195	7.228	13.204	13.204
emc	2.269	2.265	4.450	4.430	1.740	1.757	3.178	3.165
cfcomp	1.000	1.000	1.896	1.871	1.000	1.007	1.564	1.577

Table 2: Normalized execution times.

	conj				disj			
	sh_65	gr_65	sh_130	gr_130	sh_5	gr_5	sh_10	gr_10
comp	11.484	8.120	23.872	16.170	10.598	7.356	21.048	14.257
cfcomp	1.000	1.025	1.965	1.962	1.000	0.952	1.752	1.775

Table 3: Normalized compilation times.

The results in Table 2, 3, and 4 are normalized with respect to the control flow compilation case, namely with respect to conj_sh_65 for the conj benchmarks and with respect to disj_sh_5 for the disj benchmarks. Table 2 shows the normalized execution time of a query when it is executed using compile & run (comp), meta-call (call), embedded meta-call (emc), and control flow compiled code (cfcomp). Embedding the meta-call results in a substantial improvement over normal meta-call, which is of course due to the massive instruction compression. However, statically compiling the control flow seems to have an even bigger impact on the execution time: there is a speedup of up to a factor 9 over normal meta-call, and it even results in slightly faster execution than compiled code, making it the fastest approach of all four. This is mainly caused by the fact that compiled code needs extra work to set up each call as opposed to the other three approaches. Both the normal and the embedded meta-call also benefit from this fact, but there is apparently too much overhead in having to deal with the control flow at run time to be an improvement over normal compilation.

For the compiled code, the execution of the ground queries is slower than the execution of the queries with variables. This is due to the instruction merging of subsequent `putpval` and `putpvar` instructions in ilProlog, which reduces the number of emulator cycles for the benchmarks with shared variables. For the other three approaches, the kind of arguments in the calls makes no difference as they all meta-call their goals. For all four execution schemes, the execution time is linear in the number of calls as expected.

The calls in the benchmarks have only two arguments. When the number of arguments increases, the number of emulation cycles for the compile & run approach also increases , whereas the effect on the meta-call based approaches will be less important. So, we claim that when the number of arguments is higher than 2, the meta-called approaches will perform even better compared to compile & run.

	conj				disj			
	sh_65	gr_65	sh_130	gr_130	sh_5	gr_5	sh_10	gr_10
comp	1.961	2.911	3.882	5.793	1.927	2.772	4.265	5.443
cfcomp	1.000	1.000	1.961	1.961	1.000	1.000	1.890	1.890

Table 4: Normalized code size.

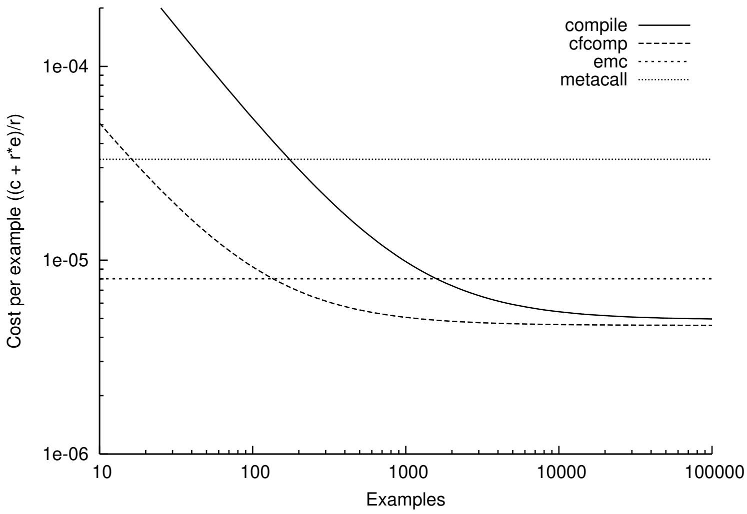


Figure 2: Cost (compilation + execution) of running disj_sh_5 on a varying amount of examples

Table 3 shows the time needed for compiling a query using the normal compiler (comp) and control flow compilation (cfcomp). The control flow compilation is up to an order of magnitude faster than normal compilation. It also generates up to 3 times less code than normal compilation, as can be seen in Table 4. The instruction merging also has an impact on the code size of the compiled queries, explaining the difference in size between the compiled ground queries and queries with variables. For the benchmarks, full compilation takes about 10 times longer than control flow compilation.

Since in the ILP setting each query is compiled only once, but is run many times (once for each example), it is interesting to get a better idea for which number of examples it pays off to do full compilation. Therefore, let $(c + r * e)$ be the total time for executing a query on r examples, with c the compilation time of the query and e the execution time for one run of the query. Computing the average over all examples $(c + r * e)/r$ gives us an estimate of the cost of evaluating a query on one example. For the benchmark query disj_sh_5, this cost is plotted in function of the number of examples in Figure 2. For the meta-call and the embedded meta-call, the function is constant, namely the execution time for running the query once. Normal meta-call is cheaper than compile & run below 200 examples, and cheaper than control flow compilation below 17 examples. The embedded meta-call is cheaper than compile & run below 2000 examples, and cheaper than control flow compilation below 150 examples. Control flow compilation always outperforms compile & run.

7 Discussion and future work

When we started with this work, we hoped that an embedded implementation of the meta-call would be competitive with the execution after a full compilation. As it turned out, this is not true: the embedded meta-call is slower by a factor two, and the slowdown became prominent when on top of conjunctions, also disjunctions were supported. We think we can still improve the embedded meta-call, but this is not our priority at the moment.

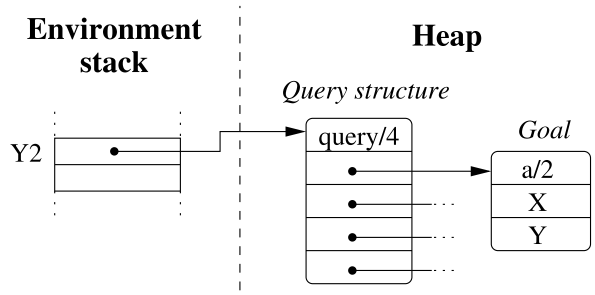


Figure 3: Memory layout for control flow compilation

Because the embedded meta-call did not live up to our expectations, we started investigating the compilation of the control flow only: it bears resemblance with structure sharing implementations of Prolog, but without some of its disadvantages. Execution of the code obtained by compiling only the control flow is competitive with the execution of fully compiled code. We have shown this for ordinary disjunctions and conjunctions, and we are confident that this result will carry over to the case of query packs: when going to query packs, the only difference in the generated code concerns the disjunction related instructions, and they are the same in both schemas. This is further supported by the fact that pack compilation takes about as long as full compilation of the corresponding disjunction. In the context of decision tree learning, it means that we can achieve a ten-fold speed-up of the pack compilation time without increasing the execution time. Since pack compilation time takes from 50 to 100% of the pack execution time, we can expect a good overall speed-up, which moreover scales nicely when the decision trees become larger.

So, our main future work is in adapting control flow compilation such that query packs [3] and other related execution mechanisms [6] are supported.

We also intend to work on improving control flow compilation: it can be seen in Figure 3 that the `arg_call` instruction needs two indirections to get at the called term. This can be reduced to one indirection with a more clever layout for the query structure. Another improvement concerns the inlining of built-ins in the query, even during control flow compilation. The built-in optimization seems impossible in the context of the embedded meta-call.

One issue mentioned in the introduction was incremental compilation: much of the complexity of incremental (full) compilation comes from having to deal with variables, i.e. variable classification, stack slot allocation and (abstract machine) register allocation. The control flow compilation never needs to deal with variables, and incremental compilation becomes a more attractive possibility again.

Acknowledgments

Remko Tronçon is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.). This work is partially supported by the GOA ‘Inductive Knowledge Bases’.

References

[1] The ACE data mining system. http://www.cs.kuleuven.ac.be/~dtai/ACE/.

[2] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990. See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[3] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence*, 16:135–166, 2002.

[4] V. S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, B. Demoen, G. Janssens, J. Struyf, H. Vandecasteele, and W. V. Laer. Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, 2002. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=38848.

[5] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[6] R. Tronçon, H. Vandecasteele, J. Struyf, B. Demoen, and G. Janssens. An Execution Mechanism for Combining Query Packs and Once-Transformations. In *Inductive Logic Programming, 13th International Conference, ILP 2003, Szeged, Hungary, Short Presentations*, pages 105–115, 2003. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=40938.

[7] H. Vandecasteele, B. Demoen, and G. Janssens. Compiling large disjunctions. In *First International Conference on Computational Logic : Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 103–121. Imperial College, 2000. http://www.cs.kuleuven.ac.be/cgi-bin-dtai/publ_info.pl?id=32065.

[8] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI, 1983.

A tag change with support for extra types and two more experiments in hProlog*

Bart Demoen † Phuong-Lan Nguyen ‡

Abstract

The original WAM specializes abstract machine code and term representation for lists, squeezes the environment stack and choice point stack into one memory area and offers support for only one atomic data type. Most Prolog systems follow the list specialization, but the control stacks are often separated and also more data types are supported at the implementation level than in the WAM. We show within hProlog the performance effect of the list specialization and the control stack choice. We also introduce in hProlog four extra basic types and show how it was done without performance loss.

1 Introduction

We assume knowledge of Prolog and its WAM based implementation. For a good introduction to the WAM [17], see [1].

Since [6], our Prolog implementation named dProlog has evolved into hProlog, which will become a back-end for HAL [5]. Over the past year, we have adapted hProlog because of HAL, and we performed some more experiments. We report here on these developments and experiments.

hProlog is WAM based. The WAM originally defined only the data types ATOM, LIST, STRUCT and REF and since heap addresses are typically a multiple of four, the encoding of these dynamic tags follows naturally. Many Prolog systems have a larger number of supported data types and some also chose to dedicate part of the heap word to information only used during garbage collection. This means that the very simple tagging scheme of the WAM is not used in practice. Another issue is that the WAM treats lists in a special way, both in the tagging scheme and in the instruction set. BinProlog on the other hand uses tag-on-data (see [16]) and it treats the ./2 functor as any other functor. We wanted to establish whether the list optimizations are really worth their while in the WAM. Section 2 reports on it.

The second experiment concerns the layout of the environment stack and the choice point stack: the WAM interleaves these stacks. Systems like XSB [13] and SICStus Prolog [11] use two separate memory areas for these two stacks; we name that the two stack model. Yap [3] on the other hand still uses the WAM layout. Yap is a particularly fast system, so we wanted to find out whether its choice for the traditional stack layout is important for its performance. hProlog was at first implemented with the two stack model, so we created a version which differs from the original one only in that aspect. We report on that in Section 3.

Finally, we come back to types and tagging schemes: in [6] we showed how dProlog - the predecessor of hProlog - has excellent performance. While converting hProlog further to a HAL back end we needed to introduce several new data types. We show how we have done this without losing performance and without restricting further the available address space for the heap: the report is in Section 4.

*This paper contains a condensed version of three internal reports which discuss each issue separately and in more depth: [12, 7] and [8] can be obtained from http://www.cs.kuleuven.ac.be/publicaties/rapporten/

†Department of Computer Science, K.U. Leuven, Belgium, bmd@cs.kuleuven.ac.be

‡Institut de Mathèmatiques Appliquées, UCO, Angers, France, nguyen@ima.uco.fr

2 Should we optimize lists ?

The list optimization is described in the original WAM and as far as we know, all implementations of the WAM indeed follow the WAM in this respect. The BinWAM [15] is an exception and it is not pure WAM in other respects as well. The list optimization consists of two *cascading* optimizations - cascading, because they are not orthogonal in practice: one does not make sense without the other.

1. the instruction set is specialized for lists (we will refer to this as LSIS)
2. the heap representation of lists is specialized (we will refer to this as LSHR)

The former has lead to instructions like *get_list* as a list-specialized version of *get_structure*: the gain is in the fact that the instruction does not need to encode as an argument the functor, because it is known to be ./2. Some WAM based implementation even have a *get_nil* instruction.

The heap representation specialization is the one that represents a list as a special tagged (LIST) heap pointer which points to two consecutive cells that contain the *car* and the *cdr* of the list: this is in line with early LISP implementations.

Specializing the heap representation for lists without specializing the instruction set as well, seems pointless: each **structure* instruction must then distinguish the LIST case from the other functors. On the other hand, specializing the instruction set **without** specializing the heap representation, makes perfect sense as we will indeed show. Note that this is in line with the experiment reported on in [6] when we implemented a tag-on-data representation scheme which does not specialize the heap representation for lists.

We list first some advantages and disadvantages of not specializing the heap representation of lists:

+ one tag becomes available for a more interesting purpose
+ support for rational trees becomes easier
+ garbage collection becomes more *safe* (see [9])
– loss of performance
– larger memory use

Note that we cannot perform the experiment just by a source to source transformation that would transform every list occurrence to another binary functor: the implementation of many built-ins have hard-wired in them that they produce or accept the specialized list representation (findall/3, =../2, sort/2 ...). So for our experiment, we need to do quite a bit of changes. We actually made two new versions of hProlog 1.8:

- hProlog_nolist which treats ./2 as any other binary functor both for instructions and for its heap representation
- hProlog_nolist(LSIS) which uses the same heap representation for ./2 as for other binary functors, but which has a List Specialized Instruction Set

2.1 hProlog_nolist

hProlog_nolist was made by changing the compiler in its abstract syntax tree building phase, and by rewriting a few internal macros and a bit of C-code. As examples:

```
#define is_list(p) tag(p) == LIST
```

became

```
#define is_list(p) is_struct(p) && *(get_struct_pointer(p)) == dot_2
```

and

```
#define make_list(p)  (((long)(p) << 1) | LIST)
```

in a C fragment like: `*x = make_list(h);` became

```
#define make_list(p)  make_struct_pointer(p)
```

and the corresponding fragment:

```
*x = make_list(h); *h = dot_2; h++;
```

Such changes are a bit tedious, but easy to get right.

2.2 hProlog_nolist(LSIS)

Introducing LSIS was done at two levels: the loader transforms some *structure* instructions with dot_2 as structure argument into the corresponding *list* instruction - we did this only for the get_structure and put_structure instructions because the other *structure* instructions occur quite infrequently. This results in code that is close to what the original compiler generates. There are small differences because the original compiler knows about the size of a cons cell. The main remaining difference however was that the original compiler generates an instruction *switchonlist_skip* - which speeds up *nrev* quite a bit - and there was no *switchonterm_skip* yet. So we had to introduce it - also by a peephole pass - and then let it be transformed to the *switchonlist_skip*.

We have in the code above used *dot_2*: at the C-level, one can think of it as a (global) variable that is initialized (at start up time) to the appropriate functor table handle for ./2. This view requires a memory access every time dot_2 is used and also in longer assembler instructions. One can alternatively make sure that the value of dot_2 is known at compile time, e.g. by running the system and printing it out and then using this value in subsequent runs. Then code will look like `*(get_struct_pointer(p)) == 179` if 179 happens to be the value of dot_2. For this to work, the handle for ./2 should be constant across runs of course. One has an interest in keeping this value small. We have chosen to make ./2 the first functor that is ever put in the functor table, resulting in the value 19. This reduces also the length of the assembler instructions needed to implement operations referring to dot_2[1]. We do not report on the effect of this trick here: it is just part of the LSIS scheme.

2.3 Benchmark results

We are interested in time and space for a set of benchmarks: these were classified roughly and a priori into benchmarks that manipulate lists a lot (the upper part of Table 1) and the other benchmarks that manipulate lists very little or in a balanced way. Performance wise, one expects to see the biggest (negative) impact of not having LSHR in the first category, and none or just a small one in the second category. Space wise, similar comments apply. For hProlog_nolist and hProlog_nolist(LSIS), we just give the relative difference to the original hProlog (which includes LSHR and LSIS) in %, i.e.

$$100 * (X(hProlog\_nolist) - X(original\_hProlog))/X(original\_hProlog)$$

where X can be time or heap size. Each benchmark was run with enough initial heap, trail ... space, so that no garbage collection or expansions were triggered.

[1]because this value is used as an immediate operand in the assembler instruction

The code size is also influenced by the choice between the three systems. The code in original hProlog and hProlog_nolist(LSIS) is virtually the same [2]. The code size in hProlog_nolist(LSIS) is about 1.3% smaller than in hProlog_nolist measured over the whole benchmark set.

The experimental evaluation was performed on a Pentium III, 500MHz, with RedHat Linux and with 256Mb RAM.

	nrev	poly_10	browse	crypt	ham	queens	reducer	zebra
nolist	+20	+14	+8	+7	+4	+3	+3	+2
lsis	+18	+9	+9	+7	+5	+1	+3	+1
	snrev	boyer	cal	send	chat	meta_qsort	sdda	comp
nolist	-21	+1	-1	0	-2	-1	-1	0
lsis	-21	0	-1	0	-2	-1	-1	0

Table 1: Performance difference of hProlog_nolist and hProlog_nolist(LSIS) with hProlog

Most of the benchmarks are classical ones. *comp* is an old version of the XSB compiler compiling itself. The upper part of Table 1 shows that the impact of abandoning LSHR is very pronounced for the *nrev* benchmark - this is as expected. In most of the other benchmarks that we judged a priori to be list-intensive (and consequently expected a bigger slowdown), we found a slowdown of 10% up to almost none.

The lower part of Table 1 shows that abandoning LSHR and in the course of doing so introducing some reasonable instruction specialization and merging, we get most often a break even. We find the figure for the benchmark *comp* most significant: it is the only real application (still of smaller than medium size) and there was no noticeable performance loss by giving up LSHR. *snrev* is a version of nrev with a binary functor different from ./2: that column shows the speed-up obtained by introducing *switchonterm_skip*.

It is clear that even with a specialized instruction set for lists, abandoning LSHR must lead to a higher memory foot print, i.e. a higher heap consumption. We have measured this to be between almost 50% for *nrev* (as expected) to 0% for *boyer* - for *comp* it was about 20%.

In [7] we also report in more detail on differences in cache behavior for the different versions.

2.4 Final comments on the LIST optimization

Changing a tagging scheme in an existing system is usually not an option - although it was quite easy in the case of hProlog. So the value of this work is clearly not in persuading people to throw away LSHR. Also, in a native code implementation, the performance difference can be expected to be larger: since there is no emulator overhead, the extra instructions executed and the extra cache misses in hProlog_nolist most certainly will show up. But it shows that by ignoring lists during the development of a Prolog system, no unreasonable penalty needs to be incurred.

The most important result for us is that on comp, the only more or less realistic benchmark, there is no meaningful speed difference between a system with LSIS+LSHR and a system without, even though the compiler uses lists itself: typical non-trivial applications indeed use lists and other terms in a rather balanced way. The other benchmarks seem to indicate that LSIS is not so effective as LSHR. This research was a necessary step for us in understanding how to use type information in hProlog: this type information is present in HAL programs. Mercury [14] has already shown the way of course, with a specialized data representation for each type. We needed to gain experience with these issues in the emulator context.

[2] indexing is influenced in unpredictable ways

3 One or two stacks ?

In this section we have again used hProlog 1.8 as a starting base.

We will use the following terminology: the *environment stack* is the stack with environments or stack frames in more traditional terminology; the *choice point stack* contains the choice points. The original WAM uses an interleaved environment/choice point stack which is named the *local* stack.

Several Prolog implementations do not use the original local stack anymore: SICStus Prolog was perhaps the first one, but neither XSB nor hProlog use a single stack for environments and choice points. Apparently, the original motivation was that locality of reference is better in a two stack model, but also that the two stack model offers easier implementation of parallel Prolog systems or tabling based on suspension/resumption of consumers as in XSB. Yap however implements the original single local stack model and Yap is very fast, if not the fastest emulator around. The natural question is of course whether this is a coincidence and indeed one of the reasons for the current work is to investigate to what extent Yap's performance could be attributed to this traditional approach to the local stack. We have consequently performed the following experiment: the implementation of hProlog 1.8 was adapted in such a way that a compile (of the C-code) time option generates a system that uses either the original WAM local stack or the (default) two stack. We will refer to these versions of hProlog as *one_stack* and *two_stack*. We will describe the adaptations in more detail in Section 3.1. We then report on the time and space performance of *_stack* on a set of benchmark programs in Section 3.2.

3.1 Changes to the hProlog machine

The text below can be understood without knowing the particularities of the hProlog variant of the WAM, but some of the quoted code only makes sense if one knows that in hProlog, environments are always *upside-down* and no environment trimming is performed.

The data structures hProlog is largely[3] re-entrant and one record - together with what it points to - captures one incarnation of the WAM as implemented by hProlog. This record - a struct in C, named *machine* - contains all WAM (and extended WAM) registers, pointers to all stacks and stack limits, the open files and the information for statistics. In *two_stack* mode, the machine contains a TOS[4] register and pointers to the boundaries of the environment and the choice point stack. In *one_stack* mode, it does not contain the TOS register and instead of the previously mentioned delimitations, the begin, the end, and the overflow limit of the single stack for choice points and environments.

One more data structure is affected by the stack decision: the choice point. In *two_stack* mode it contains the top of environment stack; in *one_stack* it does not.

The code The code for resetting the registers on backtracking differs, as in the *one_stack* mode, TOS need not be reset - and when a choice point is created, it need not be saved.

At the moment that a choice point is pushed, the top of the choice point stack must be determined. In *two_stack* mode, this is trivial: the top of the choice point stack is always the current *B*. In *one_stack* mode, we get code like:

```
if (B < E) topofcp = B; else topofcp = E;
```

[3]The program and symbol tables are global by choice; the interrupt routines (in C) need to know which machine is executing.

[4]top of environment stack

In *two_stack* mode and assuming that the top of the environment stack - TOS - is set correctly at the start of the execution, it needs updating at the *allocate* and *deallocate* instructions. At *allocate* TOS comes in correct, and is changed by

```
TOS = TOS - nrofpermvars;
```

for allocating an environment with *(nrofpermvars-2)* permanent variable slots. (the 2 represents the fixed part of the environment: the environment back pointer and the continuation pointer)

At *deallocate*, the computation of TOS must take into account environments blocked by the current top choice point:

```
if (E > B[tos]) TOS = B[tos];
          else TOS = E;
```

A similar piece of code is needed in *two_stack* mode when a *cut* is executed.

In *one_stack* mode we must compute the top of environment stack at *allocate*, since we chose not to save it in the choice points. The code is similar as above for *deallocate* in *two_stack* mode. This is the only place where the top of environment stack is computed and used in *one_stack* mode.

The code and heap garbage collectors, and the expansions of the run-time stacks were disabled: they are not called during the tests because we start all tests with an initial size that is large enough.

3.2 Time and space performance

The time and space performance was measured on a set of classical benchmark programs and a compiler compiling itself (some artificial benchmarks are reported on in [12]). Timings are reported in milliseconds: benchmarks were repeated a number of times until some reasonable total was obtained. The $\%difference$ column contains the excess of *one_stack* over *two_stack*, i.e. a negative sign reflects badly on *two_stack*. Timings were made on the same machine as in 2.3.

Table 2 shows differences of about 3% in time both ways. That hardly seems meaningful, but the $meta\_qsort$ and $queens$ are very backtracking intensive. This seems to indicate that backtracking programs benefit from having a single stack. On the other hand, $sdda$ contradicts this general conclusion.

For comparison: SICStus 3.10.0 and Yap-4.4.4 take 1755 and 1440 msecs for $comp$ respectively. SICStus follows the two stack model, Yap has single stack.

The same table contains data about the difference in space usage between the two systems: we mention the sum of the maximal usage of in the environment stack and the choice point stack in the two implementations. There are two reasons for expecting smaller numbers for *one_stack*: (1) a choice point is smaller (6 as opposed to 7 fixed entries) and (2) one can easily construct a program that allocates N stack cells for a number of environments, deallocates them and then allocates the same amount of space for choice points: in *one_stack*, this takes up N cells, but in *two_stack*, it uses at least $2 * N$: the actual worst case ratio is $(2 + 1/6)$. On the other hand, in the *one_stack model*, the cut can free the space of a choice point, but it cannot be reused immediately as in the *two_stack model*, so in that case the *two_stack* model has a smaller stack usage. The worst case for the *one_stack model* depends on the maximal number of slots for arguments in the cut away choice points. If that maximum is N, the worst case constant factor is asymptotically $(N/3 + 3)$.

Benchmark	Time			Space		
	one	two	%diff	one	two	%diff
boyer	690	700	-1.5	424	448	-5.4
browse	870	860	+1.1	5736	6139	-7.0
cal	1140	1160	-1.8	37	47	-27.0
chat	980	970	+1.0	1830	1913	-4.5
crypt	700	700	0	98	106	-8.1
ham	1400	1390	+0.7	494	542	-9.7
meta_qsort	910	940	-3.2	4892	4631	+5.3
nrev	1060	1040	+1.9	189	192	-1.5
poly_10	530	530	0	132	144	-9.0
queens_16	1160	1180	-1.7	268	287	-7.0
queens	2220	2290	-3.1	195	216	-10.7
reducer	280	280	0	794	764	+3.7
sdda	690	670	+2.9	287	303	-5.5
send	760	770	-1.3	87	112	-28.7
tak	790	790	0	166	177	-6.6
zebra	1680	1680	0	141	179	-26.9
average			-0.4			-9.2
comp	1249	1256	-0.6	12402	12111	+2.3

Table 2: Time (msecs) and space (machine words) performance: one stack against two stack

On the whole, the *one_stack* model is favorable to backtracking intensive programs.

Note that the space figures in Table 2 include the setup for the benchmarks[5].

The *one_stack* model has more chance to win space wise when the life times of choice points and environments overlap: this seems not true in more realistic programs like comp. The space comparison never shows the extreme worst case. Most often the *two_stack* model uses significantly more space: on top of that, in a realistic scenario the *two_stack* model must pre-allocate more space (roughly double) than the *one_stack* model.

3.3 Final comments on the one vs. two stack issue

The power of this contribution is in the fact that within basically the same system the two alternative stack layouts are implemented and that all other aspects of the abstract machine were kept the same: this means that any observed performance differences can be attributed to the difference between one and two stacks. The fact that hProlog has reasonable performance compared to other state-of-the-art WAM emulators, adds to the credibility of the experiment.

Folklore tells that the locality of reference is better in the single stack model and that consequently the performance should be better. The experiment confirms this, but in a weak sense: the differences are so small that they seem hardly significant. The main conclusion is that there is no good performance reason to chose one model over the other and that certainly the excellent speed of Yap cannot be attributed to its traditional stack layout.

[5]that is why the diff column for nrev is not equal to zero

4 New tags for new data types

The tag change reported on in this section, was performed within hProlog 2.3 and is meant to stay.

At some point we felt the need to incorporate two more data types in hProlog: character and string. While these can be implemented without adapting the tagging scheme, it is aesthetically more pleasing to do so and definitely from a performance point of view more desirable. We had previously also introduced attributed variables in hProlog, and we wanted to support *big integers*.

These new types were not supposed to slow down the system, neither (further) restrict the address space that hProlog in principle allows for the heap: it used to be 2 Gb on a 32-bit machine[6].

During the development of hProlog (starting with dProlog) we made one big mistake: while most data representation issues are hidden carefully, the decision to represent a free variable as an untagged self-pointer is freely used everywhere in the code, as well as the fact that a reference is an untagged pointer. This means that we could not (easily) change the representation of a reference chain and an undefined variable.

In Section 4.1 we explain the hProlog tagging scheme. The interaction with the garbage collector can be found in [8]. Section 4.4 contains the experiments and Section 4.5 finished with a conclusion.

4.1 Three tag bits and the works

We are developing hProlog almost exclusively on 32-bit machines. Pointers to the heap are aligned on a 4 byte boundary and the highest order bit of pointers returned by malloc is 0 (malloc is used for areas like the heap). This means that we can use 3 bits for the tagging. We chose to use three lower bits and to shift the word when necessary. This resulted in the following tags in bits 1 to 3:

```
REFERENCE   000 and 100 (**)          ATT         101         (*)
STRING      001         (*)           STRUCT      110         (*)
SIMPLE      010                       LIST        111         (*)
NUMBER      011         (*)
```

(**) means that the contents of the word is to be interpreted as a pointer. As usual in the WAM, a self-reference means an undef - the end of a reference chain.

(*) means that the word should be (logically) shifted to the right, and masked with $\tilde{}0x3$, to obtain a pointer [7] to a sequence of cells that contain more data. The case LIST is implemented as in traditional WAM: the pointer points to two consecutive cells on the heap and these are the head and tail of the list. In the other (*) cases, the (derived) pointer points to a *header* which also must have a tag because of the heap garbage collector.

SIMPLE: The tag SIMPLE is overloaded and means that the rest of the word encodes an atom, a character, a small integer or is a structure header (i.e. a functor descriptor) or a string header. Atom, character and small integer are *atomic* in the Prolog sense of the built-in. Let a cell have the SIMPLE tag - remember it is only 3 bits.

- If the cell's fourth bit is **zero**, the remaining 28 bits encode a small integer in the usual two's complement. This choice ensures that detecting whether some cell contains a small integer is fast.
- If the fourth bit is **one**, bits 5 to 11 (7 bits) often contain an arity, so we name this field the *generalized arity* or *genarity* for short.
  - If the genarity equals 0x7f, the cell is an atom and the remaining 21 bits denote an index in the atom table.

[6] hProlog does not restrict the size of the other memory areas

[7] [2] describes an alternative to shifting: negation of the bits can be used for one of the pointers

- If the genarity equals 0x7e, the remaining 21 bits encode a character.
- If the genarity equals 0x7d, the cell contains a STRING header, and the 21 bits encode the length of the string.
- In all other cases the cell contains a functor, i.e. the header of a structure on the heap. The 21 bits denote again an index in the functor table, where the name, the real arity and (if present) the code entry point of the functor can be found. If the genarity is different from zero, it represents the actual arity of the functor, otherwise the arity must be looked up in the functor table. It means that functors with an arity smaller than 124 (0x7d) do not need this general look-up and that covers most occurrences of functors and it allows support for very large arities.

Note that general unification and abstract machine instructions do not need to distinguish between these last 4 cases: a header is only reachable by its tagged pointer and the case ATOM, SMALLINT and CHAR need no distinction. Section 4.2 gives an overview of the representations in the form of pictures.

NUMBER and STRING: A NUMBER tagged word is a pointer to the header of either a floating point number (a C double) or a bigint. As for any tagged pointer, the pointer is extracted from the word w by

```
pointer = (((unsigned long)w) >> 1) & ~0x3
```

The same is true for LIST, FUNCTOR and ATT tagged pointers.

The header of a bigint (BIGINT_HEADER) has the value 0 and the REAL_HEADER has the value 1. At first we tried to pack more info in the BIGINT_HEADER, but later refrained from doing so mainly for simplicity reasons.

The REAL_HEADER is followed by two cells: the double. Its precision and range is as in C.

The BIGINT_HEADER is followed by a cell that encodes the length (in words) of the bigint, its sign and a bit telling whether to find the (absolute) value in the next cells or not. The latter is related to heap overflow. Execution that does not overflow the heap during a computation, always produces the value cells immediately after the information cell. For more on this issue, see [8].

A STRING tagged pointer points to a cell which is identified as a STRING_HEADER because it has a SIMPLE tag and a genarity that equals 0x7d. The remaining 21 bits indicate the length of the string in bytes: the length in words is easily derived from that. The characters are represented as in C. There is no trailing zero byte and in principle a string can contain zero bytes. It would be easy to incorporate strings with wide characters in this scheme.

4.2 Pictures that show the tags and the heap layout

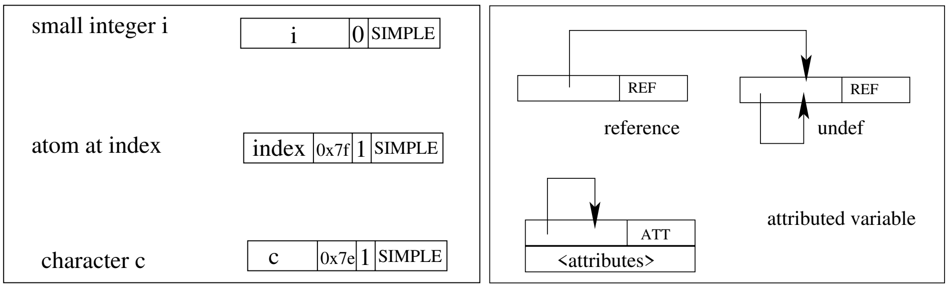
Figure 1 shows the atomic types with a SIMPLE tag and which are tagged-on-data, as well as the two kind of variables in hProlog: ordinary (Herbrand) variables and attributed variables. A reference has the same tag as a free variable.

Figure 2 shows the Prolog types which use tag on pointer: LIST, STRING, STRUCT and NUMBER. In the case of STRING, STRUCT and NUMBER, the pointer points to a header. The LIST pointer points directly to two other Prolog terms.

4.3 Code generation considerations

New abstract machine code instructions must be introduced for supporting directly the new types. For characters, these are just variants of the instructions for atoms.

The instructions for strings are similar, but one must cater for the fact that not every string has the same length. As a result, some string instructions have a variable length themselves.



(a) The SIMPLE tag with variants

(b) Variables

Figure 1: Simple tag and variables

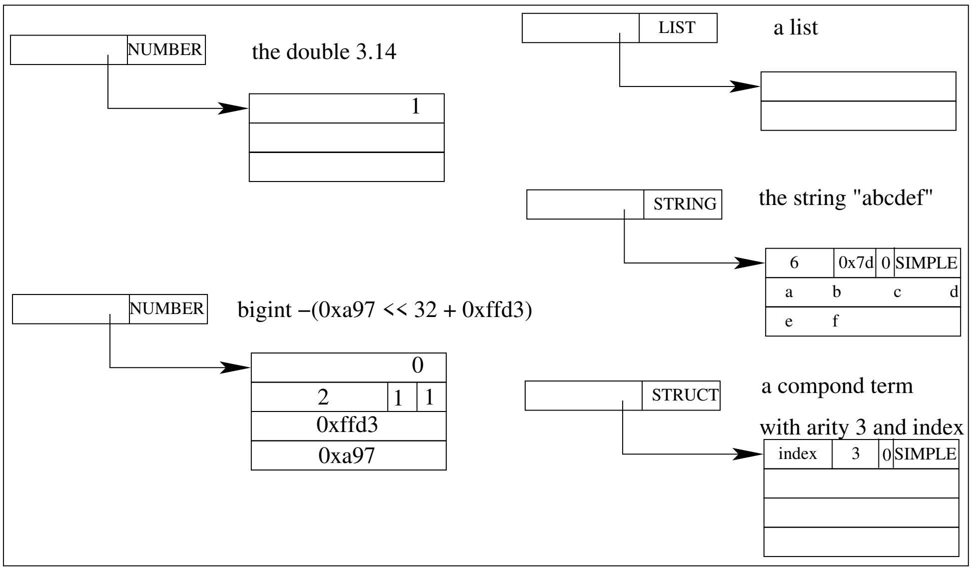


Figure 2: Other types

The instruction switchonbound [8] uses a hash table for fast access of the appropriate clause(s). It is trivially extended to also work on input and head arguments that are of the type character or string. However, in HAL - just as in Mercury - a set of facts like

```
foo(asd).
foo("hello").
```

cannot be typed correctly, because a union type with string and some other data is not possible. The same applies to other built-in types. We have therefore disabled indexing on arguments which are a mixture of values of these basic types and something else. We have not (yet) taken advantage of that by specializing the indexing instructions, but it has a detrimental effect on at least one benchmark: cal.pl contains the predicate cal_key/3 of which some facts are shown

```
cal_key( 1, 6, 1).
cal_key( 2, 2, 1).
```

[8] name inherited from XSB

```
...
cal_key(jan, 6, 1).
cal_key(feb, 2, 1).
...
```

hProlog does no longer apply indexing to such a predicate. The benchmarks show therefore timings for cal.pl and for simple_cal.pl, which is a version of cal from which we eliminated all the facts for cal_key with an atom in the first argument: they are not activated during the benchmark. While on the topic of changes to benchmarks: in sdds, we have replaced the occurrence of `"A" = [A]` by `A = 0'A` because of the introduction of strings.

4.4 Experimental evaluation of the new tag scheme

Extending the tag scheme - and the instruction set - of a system that is among the fastest around, runs the risk of slowing it down: we intended to show that the introduction of more natively supported types does not need to slow down an already fast system. We use the same set of benchmarks as in [6] and we simply compare the performance of dProlog1.0 which has the original tagging scheme with hProlog2.3.9 which has the new tagging scheme with the extra types supported: see Table 3. We also show the performance of SICStus Prolog 3.10.0 and Yap-4.4.2 because that gives additional credibility to the figures. One must keep in mind that the benchmarks are old, mostly badly written and that hProlog intends to support a super-set of Clocksin-Mellish Prolog (except for dynamic predicates).

We have previously compared the performance of our attributed variables to the SICStus Prolog ones: see [4]. The bigints in hProlog are based on Section 4.3.1 in [10], and so are the bignums in SICStus Prolog: the comparison is therefore not interesting, i.e. the results are basically the same. Mainstream Prolog systems do not support strings or characters, so a specific comparison is not possible: it is of course very easy to make benchmarks that show the superiority of supporting strings natively to supporting them as lists of ASCII codes.

benchmark	dProlog	hProlog	SICStus Prolog	Yap
boyer	*7410*	5645	6115	**5087**
browse	7552	5290	*11667*	**5270**
simple_cal	1495	**1000**	1370	*1515*
chat	807	**740**	*1007*	840
crypt	3850	**2460**	*4237*	3902
ham	977	977	*1252*	**852**
meta_qsort	*1372*	1095	1132	**1072**
nrev	6472	**6450**	*11605*	6605
poly_10	*557*	**385**	490	470
queens	1807	**1435**	*2385*	1715
queens_16	1365	**632**	*1592*	700
reducer	*5362*	3517	4130	**3382**
sdda	*460*	372	370	**340**
send	6547	**4332**	*9345*	5240
tak	1235	**837**	*1282*	1142
zebra	4210	4560	*5097*	**3485**
cal	1480	*1637*	**1357**	1510
comp	1347	**1300**	*1755*	1440

Table 3: Timings for some benchmarks

The table shows in bold the figures that are best amongst the four and the worst figure is indicated in italic. Timings are in milliseconds and the benchmarks were performed on an Intel 686, 1.8 GHz with RedHat Linux. The benchmarks were compiled without some of the optimizations that hProlog can perform (like in-lining and switch improvement).

We have singled out two benchmarks: *cal* for reasons explained in Section 4.3, and *comp*, because it is not a traditional benchmark.

Of the 16 traditional benchmarks, hProlog is the faster on 9 and never the slowest. hProlog is slower than dProlog only on zebra: the reason is that the general unification routine in dProlog does not support cyclic terms, while in hProlog it does.

hProlog is slower on the original (bad typed) version of *cal*, because it does not index any longer on a badly typed argument.

On modern hardware, the traditional benchmarks are no longer well suited for performance assessment, because they need to be repeated a million times in order to obtain timings that are close to a second. The last benchmark (*comp*) is the most realistic benchmark of all and hProlog also performs best for it.

We must end with a caveat: the choice of the version of gcc with which hProlog and dProlog is compiled is crucial to the findings. We have easy access in our department to gcc 2.95.4 and 3.0.4. hProlog runs slightly faster when compiled with 3.0.4 while dProlog benefits significantly from 2.95.4. So we have compiled each with the gcc version that gives best results. The benchmark that is most affected by the choice of compiler is nrev.

4.5 Discussion of the tagging scheme

All together, Table 3 seems to show our point: there is no need to lose performance or address space while introducing more types in the WAM. And there is room for more: we have already experimented with overloading the representation for STRINGS, so that we can have a native array type in hProlog.

5 Discussion

We have once more used hProlog as a platform for experiments. The exemplary performance of hProlog and its completeness as a Prolog system are a guarantee that the experiments and the conclusions can be meaningful. In the case of the one stack versus two stack experiment, the conclusion is that it does not really matter for performance. In the case of the list optimizations, the conclusion is that it matters only for applications using lists much more heavily than other data structures: our experience (independent of the benchmarks) is that this is rare. The conclusions are not surprising, but it is nice to have an experimental verification of what is intuitively believed. The final experiment - a new tag scheme allowing more native data types - is the most important to us, because we cannot dodge the issue of supporting natively more data types. So it is all the more important that we have achieved this without performance loss. We acknowledge that performance is secondary to features, but it is our duty to search techniques which do not trade one for the other. We believe we have succeeded.

Acknowledgments

Part of this work was conducted while the first author was a guest at the Institut de Mathematiques Appliquées of the Université Catholique de l'Ouest of Angers, France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for his work on the ilProlog compiler used within hProlog, and the referees for relevant comments that helped improve the paper. This work was partly supported by project G.0144.03 funded by the Fund for Scientific Research - Flanders,Belgium (F.W.O. - Vlaanderen).

References

[1] H. Ait-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] V. S. Costa. Optimising Bytecode Emulation for Prolog. In *Proceedings of PPDP'99*, volume 1702 of *LNCS*, pages 261–277. Springer-Verlag, Sept. 1999 See also http://www.ncc.up.pt/~vsc/Yap/.

[3] L. Damas, V. Santos Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*, 1989.

[4] B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002. unpublished.

[5] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.

[6] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[7] B. Demoen and P.-L. Nguyen. What if [a] really equals .(a,[]) ? Report CW 351, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Oct. 2002.

[8] B. Demoen and P.-L. Nguyen. Supporting more types in the wam: the hprolog tagging scheme. Report CW 366, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Aug. 2003.

[9] B. Demoen, P.-L. Nguyen, and R. Vandeginste. Copying garbage collection for the WAM: to mark or not to mark ? In P. Stuckey, editor, *Proceedings of ICLP2002 - International Conference on Logic Programming*, number 2401 in Lecture Notes in Computer Science, pages 194–208, Copenhagen, July 2002. ALP, Springer-Verlag.

[10] D. J. Knuth. *Semi-numerical Algorithms*. Addison-Wesley, 1981.

[11] T. I. S. Laboratory. *SICStus Prolog User's Manual Version 3.7.1*. Swedish Institute of Computer Science, Oct. 1998.

[12] P.-L. Nguyen and B. Demoen. Interleaving or separating environments and choice points in the wam. Report CW 364, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Aug. 2003.

[13] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of SIGMOD 1994 Conference*. ACM, 1994.

[14] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.

[15] P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Russian Conference on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.

[16] P. Tarau and U. Neumerkel. A novel term compression scheme and data representation in the binwam. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 73–87. Springer-Verlag, Sept. 1994.

[17] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

73

Controlling Code Expansion in a Multiple Specialization Prolog Compiler

Michel Ferreira   Luís Damas
DCC-FC & LIACC, University of Porto,
Rua do Campo Alegre, 823 - 4150 Porto, Portugal
{michel,luis}@ncc.up.pt

Abstract

In this paper we present the work developed in the u-WAM compiler to control the code growth that results from the multiple specialization of a program. Few attention has been devoted to this issue, but it is extremely important for the compilation of real programs. We show how the u-WAM system allows performing *partial multiple specialization*, solving the important problem of scalability of existing global analysis frameworks. We describe a WAM level profiler which guides the selection of predicates for specialization and show that the distribution of time per predicate in large programs is adequate for partial multiple specialization, allowing largely improving the efficiency of a program by multiply specializing a small number of well chosen predicates. We also show how code growth can be further improved by collapsing versions of a predicate that do not introduce new optimizations, or that introduce only negligible optimizations, through a non-strictly identical collapsing of versions.

1 Introduction

The u-WAM Prolog compiler [5] implements multiple specialization of Prolog predicates based on an abstract interpretation of WAM [14] code [6]. The most important aspect of this prototype system is the fact that analysis is much more intra-procedural and less inter-procedural than other analysis implemented using abstract interpretation over the Prolog source code. This is due to the availability of low-level information at the WAM level.

A very important aspect when implementing a multiple specialization strategy into a compiler is to develop an efficient way of controlling code expansion. Multiple specialization unfolds the code of a predicate in a number of codes implementing specialized versions of the predicate. As a result, code growth is particularly problematic in a multiple specialization compiler.

Keeping analysis intra-procedural presents the advantage of allowing to apply the process of analysis and specialization to a number of isolated predicates within a program. The interface between non-specialized predicates and specialized predicates is done via a special *most-general-version* which, in the case of recursive predicates, directs execution to the specialized versions. This *partial multiple specialization* allows to control code growth while obtaining important speed-ups in the compilation of large programs, as long as the time relevant predicates are selected to be multiply specialized.

Program profiling comes immediately to mind as the source for information for choosing predicates. However, Prolog profiling is more complex than profiling of simpler control-flow languages. We address Prolog profiling and describe our implementation of a WAM level profiler.

Another important aspect in the code expansion control of a multiple specialization implementation, is obtaining an optimal set of specialized versions within a predicate. Our WAM level multiple specialization is particularly advantageous on this aspect, as the implementation differences between versions are clearly visible at this level. We explain how a *collapsing* of non-strictly identical versions is performed to further reduce code expansion.

2 Partial Multiple Specialization

The number of versions generated for each predicate is a function of the arity of the predicate and of the analysis domain complexity. The elements of the analysis domain are of two types: elements which actually lead to optimizations and elements used to propagate analysis information. If d is the number of elements of the domain and n is the number of elements which do not lead to optimizations, then, in the worst possible case, the number of versions generated for a predicate will be:

$$P_{versions} = (d - n)^a$$

where a is the arity of the predicate. This is a wildly pessimistic number of versions as, due to the structure of the domain, only a pathological predicate could achieve such a variety of activation patterns. It is, though, an indicator of the order of growth of a multiple specialization implementation, revealing its exponential nature. In real-size programs only the most critical predicates of a program must be multiply specialized, performing *partial multiple specialization* instead of total multiple specialization.

Traditional global dataflow analysis analyze programs considering all its predicates and all the interactions between them, as a whole. Altering one predicate normally requires re-analysis of the entire program. Most works on global dataflow analysis of logic programs typically assume that the entire program is available for inspection at the time of analysis. Consequently, it is often not possible to apply existing dataflow analyses to large programs, either because the resource requirements are prohibitively high, or because not all program components are available when we wish to carry out the analysis. This is especially unfortunate because large programs are typically those that stand to benefit most from the results of good dataflow analysis. Research on global analysis of logic programs is focusing on *incremental* or *modular* analysis, in order to overcome this problem [12, 9]. Essentially, this involves the extension of the fix-point algorithms of the generic analysis engines to support incremental/modular analysis.

In most analysis frameworks it is, therefore, impossible to select just a part of a program for analysis and to produce results that are *correct* for the entire program. Because we extract information from a predicate in a local basis, it becomes possible to analyze predicates individually, selecting a number of predicates for analysis, and performing multiple specialization for them. This is one of the most important aspects of our approach, as it does not have the scalability problems of traditional global analysis, it can be applied to real programs, and if the predicates are well chosen performance can be significantly improved.

Performing partial multiple specialization in the u-WAM system is simple. The WAM code of a predicate is executed by an abstract emulator that modifies a local (predicate scope) *State* structure according to the WAM instruction being abstracly executed. A `call` or `execute` instruction use the *State* information to construct the calling pattern. If a calling pattern is detected for a predicate that is not to be multiply specialized, then the calling pattern is replaced by the generic pattern. The generic version of the called predicate is (abstractly) executed with an empty *State*, as opposed to a *State* initialized with the information from the calling pattern in a predicate to be multiply specialized. Figure 1 presents the abstract meaning of `call(`P/N`)`. The *AbstractEmulate* procedure executes each WAM instruction producing the *State* modifications. It also outputs a new specialized WAM instruction of the current instruction being executed, for instance a `get_list`$\_lst_{ndrf}$`(`A_i`)` when the current *State* determines that A_i is a dereferenced list. In the end of the abstract emulation of a predicate its specialized code has been generated.

$\boxed{\texttt{call}(P/N)}$

if $(P/N \in \mathit{Selected\_Preds})$ $P'/N \leftarrow \mathit{BuildCallingPattern}(P, N, \mathit{State})$
else $P'/N \leftarrow \mathit{BuildGenericPattern}(P, N)$
$WamAnalyze(P'/N)$
$ExitP'/N \leftarrow ReadExitTypes(P'/N)$
$NewState \leftarrow UpdateState(State, ExitP'/N)$
$AbstractEmulate(NextInst, NewState) \rightarrow outcode_1$
$outcode \rightarrow \texttt{call}(P'/N), outcode_1$

Figure 1: Abstract meaning of `call(`P/N`)` for partial multiple specialization

Partial multiple specialization tends to be goal-independent. At some point in the call graph, some calling patterns will be promoted to their most general version, and the subsequent calling patterns will be derived as goal-independent. On partial multiple specialization, the predicates which are not selected for being multiply specialized *and* do not call predicates which are to be multiply specialized, do not have to be abstractly executed. Efficiency is improved by separating these predicates and generating independently non-specialized code for them.

Partial multiple specialization fits naturally with our intra-procedural analysis and goal-independent specialization, and is easily implemented. However, partial multiple specialization can only yield good results if the predicates selected for multiple specialization are correctly chosen. The next section addresses the problem of selecting the time relevant predicates to be multiply specialized.

3 Profiling Prolog Programs

To optimize partial multiple specialization we must be able to correctly choose which predicates to unfold. We want to generate specialized versions for the predicates which consume more time, where optimizations will result in important speedups in the overall program execution.

To measure the most important parts of a program, profiling techniques have been used in several programming languages to collect information during the execution of the program. The information provided by the profiler can be used for other purposes than just improving speed of execution, such as program debugging and code optimization[2]. For a developer a simple and economical way of obtaining efficient programs is to write the program, obtain execution profiles for it on representative input data, and then rewrite the most often executed parts in an optimized way. The program can be profiled again, returning new information that can be used again in an iterative optimization process. In our case, where the program is automatically transformed to become more efficient and where that transformation has important consequences on code growth, the information from execution profiles is much more useful, even essential. Totally unfolding a 1000 lines program with high-arity predicates, causes code to enlarge to unreasonable limits, eventually leading to slow-downs rather than speedups due to caching problems. However, it happens very often that a 1000 lines Prolog program spends 90% of the time in just a few predicates. If the profiler detects those few predicates and multiple specialization is done just for those predicates, then performance can be significantly improved with very low impact on code growth.

The implementation of profilers for traditional languages is simple and well understood. There are two types of profilers: *time profilers* and *count profilers*. Time profilers give information about the time spent in each procedure or piece of code. This is the most useful information to the developer in order to improve program efficiency. It is difficult, though, to accurately time profile a program, because the overhead of the timing instructions is large. This overhead, together with the coarse grain of system clocks, can mislead the time estimate of fast procedures. Another problem with the overhead of the timing instructions is that the candidate programs to profiling and consequent optimization are programs which already take long to execute, possibly becoming prohibitive when being profiled.

A solution for those problems is based on statistical sampling of the program execution. An interrupt is generated on a regular or random time cycle, and we annotate the routine or part of the code where the program counter was at that time. This idea can be easily applied to Prolog programs, measuring the time spent inside each predicate definition. The overhead is small and there is no need to change the program code. Errors may be, however, relatively large[8].

Count profilers give information about the number of times a procedure or piece of code is executed. Execution counts do not give time information directly. However, if we know how much time an instruction takes to execute the count information can be converted to time information. The profiler we have implemented is a count profiler which converts the count information to relative time information. Count profilers have some advantages over time profilers, because they are exact and are able to sketch the execution path of the program.

3.1 The Control-Flow of Prolog Programs

Classical languages have a simple unidirectional flow of control, and the sequence of the program instructions is identical to the sequence of execution. In Prolog, however, the execution has two possible directions, forward and backward. In this case the sequence of the program execution is not identical to the sequence of the program instructions. For example, a possible execution sequence for the following program

$$h \texttt{ :- } p_1, p_2, p_3.$$

can be

$$h, p_1, p_2, p_1, p_2, p_1, h \text{ (fail)}$$

This more complex mechanism of flow of control introduces some interesting problems, concerning the management of timing in the presence of backtracking and failure. These problems have been addressed and solutions proposed by Debray in [4]. Essentially, care must be taken as Prolog procedures can be entered via *call* or *redo* ports, and exited via *exit* or *fail* ports, following the box-model of Byrd [1]. In [10], Matos describes an interesting Prolog count profiler which displays valuable information, by defining four count values (called *currents*) based on these four ports. Besides showing the values of these currents, the profiler also evaluates *intrinsic* predicate characteristics based on a matrix model, that detail the origin of each current value.

3.2 WAM-Level Profiling

Prolog profilers such as [7], [4] and [10] are implemented over the Prolog source code, rewriting the program clauses of the predicates to be profiled to call special predicates before and after executing the call. Those special predicates are responsible for the timing or counting of predicates.

Our approach is different, as the profiling is performed and implemented at the WAM level. A first advantage of such approach is that the compiler optimizations (such as clause indexing) that could mislead the true flow of control on the Prolog source profiling implementations, are explicit in the WAM code.

A second reason to implement a profiler at the WAM level is the higher level of detail introduced by WAM instructions in the codification of a predicate, leading to more accurate conversion of count information to time information, as, due to the low-level of the WAM instructions, we can more or less fairly estimate how much time each instruction takes to execute.

A third reason is that the WAM code is more similar to the code of traditional programming languages, with a simpler control-flow and the traditional problems of Prolog profiling disappear.

A last reason is to perform profiling at a level that can be used with the process of multiple specialization, contributing to obtain an optimal set of versions for a particular predicate.

The implementation of our WAM profiler is simple but effective. The basic idea is to give a certain weight to each WAM instruction. The weight is meant to represent a relative time that the instruction takes to execute. We obtained this weight by inspecting the implementation code of each instruction, counting the number of assignments, comparisons and operations performed. The values of these weights can be seen as an equivalent measure to the CPU instruction cycles each WAM instruction would take to execute.

Each instruction increments a counter with its weight. The counter can be a single global counter translating, upon program termination, the total execution time, or it can be a number of counters, one per predicate, translating the time spent in each predicate, or it can be a counter per instruction type, for instance, grouping instructions in indexing instructions, control instructions, unification instructions, etc, giving information about particular characteristics of the program. Such characterization is important because, in Prolog, performance depends on different aspects, such as data structure implementation for some programs, and control for others. We are basically interested in having a counter per predicate, to order predicates by their relevance.

The single global counter can be used to check the accuracy of the count values assigned to each WAM instruction. The proportionality of the real time a benchmark takes to execute and the global count value obtained should be the same across different benchmarks. On the set of benchmarks that we have tested, the error obtained was smaller than 7%.

A problem with giving a fixed weight to a WAM instruction is that the complexity of the instruction varies, depending on the input. For instance, a large number of WAM instructions have an intrinsic dereferencing operation. This dereferencing operation is of variable complexity, depending on the chain length to be dereferenced. Some other instructions operate either on write or read mode, and their complexity varies depending on this execution mode. These variations are of small relevance [13], though. There is, however, an important problem when giving a fixed weight to a WAM instruction, and this problem has to do with unification instructions, namely unification between non-variable terms. The problem is that the terms to be unified can have very different depths and arities, and such depth and arity linearly increases the complexity of the unification instructions. Moreover, unification is everywhere in Prolog programs, making crucial the correct measurement of such operation. WAM instructions like `get_value`(X_i) and `unify_value`(X_i) call a general unification routine, *Unify*, which has a varying complexity. In the cases where one of the terms is unbound the unification is reduced to making the unbound term point to the other. This has a fixed cost that does not introduce errors in our count profiling. The problem is the unification of two compound terms. This unification is made recursively and, in the case of structures, this recursion is iterated in the number of arguments. The size of such recursion and iteration must be considered when profiling the program to obtain accurate results. Therefore, the instructions

```
[p(1,1),p(2,5),p(3,8),p(4,6),p(5,3),p(6,7),p(7,2),p(8,4)]
time : 610

Profile information:

   Characterization             Time/predicate

Get              --> 29%      exe_1/0     --> 00.00%
Unify            --> 39%      q/0         --> 00.00%
Put              --> 04%      q8/0        --> 00.00%
Call             --> 13%      q/2         --> 00.81%
Choice-point     --> 09%      perm/2      --> 09.32%
Indexing         --> 06%      safe/2      --> 09.66%
                              test/2      --> 10.25%
                              nd/2        --> 12.34%
                              sel/3       --> 15.00%
                              pair/3      --> 42.61%
```

Figure 2: Profiling report of the `queens8.pl` benchmark

`get_value(`V_i`)`, `unify_value(`V_i`)`, and `unify_local_value(`V_i`)` cannot be accurately count profiled in the same manner as the other instructions because they depend on the size of the input.

To accurately count profile these instructions we modified the *Unify* routine to return a number representing the size of the recursion and iteration of its execution. The weight of the WAM instruction is then calculated based on the returned value, translating the size of the input. If we want more accuracy on our profiling information, the same scheme can be implemented on the instructions which operate in write or read mode, returning different values according to the execution mode.

3.3 Information from Profiling

We use the profiling information in three distinct situations: for program characterization, for selection of predicates for multiple specialization and for elimination of versions of the multiply specialized program.

3.3.1 Using the profiler for program characterization

We use the profiler reports to characterize a particular program. Such characterization is important because in Prolog performance depends of different aspects, such as data structure implementation for some programs, and control for others. Interpreting the speed-up results obtained on a particular benchmark is easier if we previously characterize the program, evaluating if it relies more in unification than in choice-point management, etc. Dividing WAM instructions in 6 blocks, *Get*, *Unify*, *Put*, *Call*, *Choice-point*, and *Indexing* instructions, and using a counter for each of these types of instructions gives the emphasis of a particular program in each of these aspects.

In such profiling, the execution of each WAM instruction increments the respective counter with the weight which represents its relative execution time. The report printed translates the time percentage spent in each of these aspects, and is illustrated in the left column of Figure 2 for the *queens8* benchmark.

From this report we observe that *Call* instructions represent 13% of the execution time. No gains are expected on the specialized implementation of this percentage of the program. Also, more improvements are expected on the unification part of the program, *Get*, *Unify* and *Put*, which represent 72% of the execution time, than over *Choice-point* and *Indexing* instructions, that only profit from a determinism improvement. We present in Table 1 the characterization profile for the benchmark set we have used. This is the same set we have used in [6].

Program	Predicates	Unification			Control		
		Get	Put	Unify	Call	CP	Index
qsort	4	25%	6%	40%	9%	13%	8%
query	6	39%	14%	1%	22%	9%	15%
derive	7	27%	4%	35%	11%	13%	10%
zebra	7	27%	1%	36%	11%	25%	0%
serialise	8	30%	3%	39%	13%	8%	7%
mu	9	26%	1%	39%	8%	18%	8%
fast_mu	9	29%	9%	31%	10%	18%	3%
crypt	10	25%	12%	44%	9%	2%	8%
meta_qsort	11	16%	8%	22%	19%	23%	12%
queens_8	11	29%	4%	39%	13%	9%	6%
nreverse	12	28%	2%	49%	9%	0%	12%
prover	15	24%	7%	31%	15%	14%	9%
browse	19	29%	4%	32%	12%	16%	7%
boyer	29	20%	12%	5%	26%	28%	9%
sdda	42	21%	5%	33%	15%	24%	2%
nand	78	28%	6%	27%	16%	18%	5%
chat_parser	158	28%	10%	26%	16%	14%	6%

Table 1: Characterization of the benchmark programs

The first column indicates the number of predicates of the program. This is the number of predicates after syntactic transformations, like *if-then-else* constructs elimination, have been performed. The smallest benchmark, *qsort*, has only 4 predicates and the largest, *chat_parser*, has 158.

The remaining columns present the results of the characterization profile for each benchmark. The percentage of the execution time in each group of instructions, *Get*, *Put* and *Unify* for unification, and *Call*, *Choice-Point* and *Index* for control, is given for each benchmark.

This table shows important differences between the programs of this benchmark set. These differences must be considered when analyzing the speed-up results obtained in the multiple specialization of these programs, as, for instance, unification instructions benefit more with specialization than control instructions.

3.3.2 Using the profiler for selection of predicates

The main task of the profiler is to give the relevance of each predicate in the total execution time of the program. For this purpose we use a counter per predicate, and WAM instructions increment the counter of the predicate to which they belong with the weight which represents its relative execution time. The report printed translates the time percentage spent in each predicate of the program, and is illustrated in the right column of Figure 2 for *queens8*.

From this report we observe that the predicate *pair/3* is the critical predicate, being responsible for more than 42% of the total execution time. The 4 most important predicates represent over

80% of the total execution time. This information is important because multiple specialization of the 4 most important predicates can significantly improve the performance of the program, while predicates like *q/2* or *perm/2* only represent a very small percentage of the execution time, and thus should not be multiply specialized.

The goal of our profiling is to choose a *reasonably small* number of predicates to multiply specialize that represents a *reasonably large* percentage of the execution time of the program. The ratio between the first and the second *reasonably* is determinant in the success of our multiple specialization strategy, and depends on the time per predicate distribution of Prolog programs. It is well known that, in traditional programming languages, programs spend the great majority of time in just a few cyclic functions. Our measurements show that this also happens with Prolog predicates, and this result is very useful for partial multiple specialization.

Considering the size differences of the benchmarks, we analyzed the profiling information with respect to the *percentage of the number of predicates*, using 10 measuring points from 10% to 100%, in order to reach a meaningful average information. This is summarized in Table 2.

Program	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
qsort	36.5	73.3	73.3	95.8	95.8	95.8	99.8	99.8	100	100
query	35.4	35.4	69.0	69.0	84.8	99.5	99.5	100	100	100
derive	86.2	86.2	89.7	93.1	96.1	96.1	98.8	99.9	99.9	100
zebra	52.6	52.6	79.0	96.4	100	100	100	100	100	100
serialise	49.6	73.5	73.5	85.8	91.3	96.4	99.5	99.5	99.8	100
mu	26.0	50.7	71.0	84.2	91.4	91.4	96.0	99.6	100	100
fast_mu	56.1	80.8	91.4	97.2	98.8	98.8	99.2	99.6	99.8	100
crypt	57.9	89.8	94.6	97.7	98.5	99.2	99.6	99.9	100	100
meta_qsort	41.0	75.4	83.8	89.5	97.6	100	100	100	100	100
queens_8	42.6	57.6	70.0	80.2	99.2	100	100	100	100	100
nreverse	91.2	99.8	99.9	100	100	100	100	100	100	100
prover	60.0	74.2	93.3	96.4	99.5	99.7	99.9	99.9	100	100
browse	95.0	99.0	99.5	99.7	99.9	99.9	100	100	100	100
boyer	87.0	97.8	99.4	99.8	100	100	100	100	100	100
sdda	76.0	89.0	95.7	98.8	99.8	100	100	100	100	100
nand	65.2	83.9	90.2	94.3	97.7	99.5	99.9	100	100	100
chat_parser	62.4	77.2	86.2	92.1	95.7	99.4	99.4	99.9	100	100
Average	60.0	76.2	85.9	92.4	96.8	98.6	99.5	99.8	100	100

Table 2: Percentage of execution time for varying percentages of most relevant predicates

On average 60% of the total execution time is concentrated on 10% of the predicates. Multiply specializing this small part of the program can significantly improve the efficiency of the program. Multiple specialization could be taken further, and applied to 20% of the predicates that, on average, account for more than 75% of the execution time. If performance is critical and the program being considered is not large we could select the 30% most important predicates, where almost 90% of the total execution time is present. Further than that only marginal improvements would be achieved, or the code growth could even introduce some slow-downs due to caching problems.

Notice that the results from Table 2 show an increasing trend as the programs become larger. Considering the last 3 programs which have more than 40 predicates, the percentage of the execution time on 20% of the predicates is on average 83.4%. Considering the first 7 programs, which have less than 10 predicates, this percentage is 64.6%.

3.3.3 Using the profiler for elimination of versions in the multiply specialized program

Because the profiler operates over WAM code, it can also be applied to u-WAM code, i.e., to the multiply specialized program. The specialized predicates created, formed by the pair *(predicate name/calling pattern)*, do not exist at the source Prolog level but exist at the (u-)WAM level. Performing profiling on a counter per predicate basis over the multiply specialized code, as for predicate selection, gives the relevance of each specialized version. These reports may show that some specialized versions are very little used, or even not used at all, and can thus be eliminated. This elimination, or *collapsing*, is explained in the next section.

4 Collapsing Versions

The number of versions of a predicate clearly influences the size of the codification of that predicate. The different versions exist because they allow different optimizations to be performed. Normally the versions are not disjunctive and are totally contained in a more general version. In a goal-independent analysis there is always a most general version of a predicate in which the specialized versions are contained. Even in goal-dependent analysis starting with a declared entry-point we commonly obtain a most general version of a predicate which contains all the other specialized versions. Finding the optimal set of specialized versions for a predicate is a complex problem, that must balance the benefits from each version optimizations with the disadvantage of the overall code growth. This problem has been addressed in [15] with an algorithm based on the notion of minimal function graphs, and in [11]. Ideally, in terms of codification, every version which introduces a new optimization should be created. Due to code growth and consequent caching problems, such rule is not valid on practice.

In our specialization process versions are created prior to knowing which optimizations will they allow, and the final set of versions is obtained by a collapsing process. When encountering a `call` or `execute` instruction, the activation pattern is calculated for the predicate being called (if it is to be multiply specialized) based on the information present on the *State* structure that analysis maintains. If the activation pattern is new then a new predicate version will be created for it, regardless of its WAM code and the optimizations the activation pattern will allow. It may happen that the optimizations allowed will be exactly the same of a previously created version, but that will only be detected in the end of the code specialization process. In such a case, collapsing the versions that allow the same optimizations into a common version benefits code size and preserves program efficiency.

The collapsing of versions occurs at three different levels in the process of specialization: in the *widening* of the activation patterns in the `call` or `execute` instruction; after the code has been generated; and after execution has been profiled.

4.1 In the *Widening* of the Activation Patterns

The *State* structure maintains information about the composition of compound registers. When the activation pattern is calculated this compound representation is converted, through *widening*, to a single atomic type, collapsing a possibly infinite number of representations into a single one.

The domain complexity greatly influences the collapsing of versions at this level. For instance, the first analysis domain of our system had only a single *list* type to represent every possible compound term whose functor was `./2`. Our current domain has more elements which represent the type *list*, namely $list_{ndrf}\_of\_int_{ndrf}$ (a dereferenced list of dereferenced integers), considering the composition of the term sub-terms. Whereas with the first domain all *list* terms are reduced into a single type, with more complex domains the reductions are weaker, reflecting on the overall collapsing of versions.

4.2 After Code Generation

At this level we compare the u-WAM code of each version of a predicate. Different activation patterns can lead to exactly the same implementation code.

Comparing the u-WAM code of the different versions of a predicate is equivalent to compare the set of optimizations of each or-record of the predicate, as is done by Puebla in [11], in the same context of collapsing versions. In [11], the algorithm of minimization of versions receives as input the set of table entries (or-records) computed during the analysis of the program, together with the set of optimizations allowed in each or-record. Such set of optimizations is calculated after analysis and prior to the execution of the minimizing algorithm. The output of the algorithm is a partition of the or-records for each predicate into equivalence classes. For each predicate in the original program as many versions are implemented as equivalence classes exist for it.

Two versions of a predicate that allow the same set of optimizations cannot be blindly collapsed, since they may call a same predicate with different sets of optimizations allowed and thus the two versions must be kept separate to allow the implementation of all possible optimizations.

Collapsing at the WAM level has some advantages. First, the set of optimizations allowed in each version is directly visible in the u-WAM instructions. Second, as the pair predicate name/calling pattern forms a distinct predicate at the WAM level, the comparison between the u-WAM code of the different versions of a predicate will fail, if they call a same predicate (at the source Prolog level) with different calling patterns. However, we might be interested in collapsing versions which only differ in the calling patterns of the `call` and `execute` instructions. We can collapse these versions if the distinct versions of the called predicates can also be collapsed. Consider the example program scheme of Figure 3, where p_1,p_2 and q_1,q_2 are two versions of the same predicates.

p_1: `get_...` `...` `put_...` ... `execute(`q_1`)`	p_2: `get_...` `...` `put_...` ... `execute(`q_2`)`
q_1: `get_...` `...` `put_...` ... `execute(`r_1`)`	q_2: `get_...` `...` `put_...` ... `execute(`r_1`)`

Figure 3: Example for collapsing of versions

We try to collapse p_1 and p_2. Assume that the u-WAM instructions of the two versions are identical except for the `execute` instructions, which call two different versions, q_1 and q_2. We can collapse p_1 and p_2 if q_1 and q_2 can be collapsed, which is the case if their u-WAM instructions are identical. If there was a difference between the u-WAM instructions of the two versions of q then p_1 and p_2 could not be collapsed, in order to allow reaching the optimizations of the versions of q.

We generalize the process of comparison to allow some degree of difference between versions of a predicate. If the difference is small, like between a `get_list`$\_lst_{ndrf}$`(`A_i`)` (A_i is a dereferenced list) and a `get_list`$\_nv_{ndrf}$`(`A_i`)` (A_i is a dereferenced nonvar) then it can be interesting to reduce code growth by collapsing the versions, ignoring the minor optimization. Collapsing non-identical versions introduces a problem that did not exist with versions with exactly the same code. The collapsed version will have just one instruction for each two possibly distinct instructions compared. This instruction is from one of the versions, and it may introduce a non-valid optimization in the other version. The solution for this problem is to generate the code for the collapsed versions based on an instruction level calculation of the least-upper-bound of the two u-WAM instructions.

In Figure 4 we present the algorithm which checks if two versions of a predicate can be collapsed.

With a threshold value of 0, every version which introduces a new optimization will be implemented. With higher values of threshold we can achieve a compromise between code size and efficiency.

```
procedure can_collapse(P_a, P_b, Diff)
    if (collapse_flag(P_a, P_b) = True) then return(True);
    else
        begin
            collapse_flag(P_a, P_b) = True;
            foreach WAM_instruction(W_i) of P_a, P_b do
                begin
                    if (P_a(W_i) = call(Q_c) and P_b(W_i) = call(Q_d)
                        and can_collapse(Q_c, Q_d, Diff) = False)
                        then
                            begin
                                collapse_flag(P_a, P_b) = False;
                                return(False);
                            end
                    if (P_a(W_i) = execute(Q_c) and P_b(W_i) = execute(Q_d)
                        and can_collapse(Q_c, Q_d, Diff) = False)
                        then
                            begin
                                collapse_flag(P_a, P_b) = False;
                                return(False);
                            end
                    Diff = Diff + difference(P_a(W_i), P_b(W_i));
                end
            if (Diff < Threshold) return(True);
            else return(False);
        end
end can_collapse
```

Figure 4: Collapsing checking algorithm

4.3 After Profiling

Although a determinate specialized version has a much more simplified code than its more general versions, and thus cannot be collapsed using the previous criteria, it is often the case that it is not executed a sufficient number of times to become a relevant version. This is where the profiler is useful when applied to the already specialized program.

Although the profiler gives information about the execution time percentage of *predicates* and the created specialized versions do not exist as predicates at the Prolog level, because the profiler operates at the WAM level and at this level the pair (predicate name, activation pattern) defines a predicate, the output of the profiler gives information about the relevance of each specialized version.

Thus, profiling a program after multiple specialization has been performed gives relative time information about each specialized version. Using this output information, we can replace the specialized versions which are executed very few times by the most general version of the predicate, resulting in insignificant performance loss and valuable code size reduction.

5 Performance of Partial Multiple Specialization

Table 3 gives, for the benchmark set, the average code growth factor and execution speed-up of the u-WAM system over the `wamcc` system [3], in which u-WAM is based. We present the values for the specialization of 20% of the predicates and for 100% of the predicates.

	`wamcc 2.22`		u-WAM	
Percentage	Object Size	Execution Time	Object Size	Execution Time
20%	1	1	1.57	1.50
100%	1	1	9.44	1.91

Table 3: Comparing average code size and execution time of `wamcc` and u-WAM

Multiple specialization of 20% of the predicates allows a speed-up of 1.5, while code size increases by 1.57. Considering total multiple specialization, the speed-up increases by 1.91, while code size increases by an unreasonable 9.44 times. The increase in compilation time is even larger. These results clearly show the importance of partial multiple specialization.

6 Conclusions

In this paper we have presented the work developed in the u-WAM compiler to control the code growth that results from the multiple specialization of a program. Few attention has been devoted to this issue, but it is extremely important for the compilation of real programs. Partial multiple specialization, though of very simple implementation in our system, solves the important problem of scalability of existing global analysis frameworks. We have also demonstrated that the distribution of time per predicate in large programs is adequate for partial multiple specialization, and allows largely improving the efficiency of a program by multiply specializing a small number of well chosen predicates.

Code growth can be further improved by collapsing versions of a predicate that do not introduce new optimizations, or that introduce only negligible optimizations, through a non-strictly identical collapsing of versions.

A future improvement will be the selection of predicates for multiple specialization that are not time relevant, but that allow, through its multiple specialization, reaching new calling patterns of time relevant predicates.

References

[1] L. Byrd. Understanding the control flow of PROLOG programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138, 1980.

[2] P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 1991.

[3] P. Codognet and D. Diaz. WAMCC: Compiling Prolog to C. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 317–332, Cambridge, June 13–18 1995. MIT Press.

[4] S. K. Debray and D. S. Warren. Automatic mode inference for prolog programs. *The Journal of Logic Programming*, 5(3):78–88, September 1988.

[5] M. Ferreira. *Advanced Specialization Techniques for the Compilation of Declarative Languages.* PhD thesis, Faculdade de Ciencias da Universidade do Porto, 2002.

[6] M. Ferreira and L. Damas. Wam local analysis. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages, 5th International Symposium, New Orleans, USA, January 2003*, volume 2562 of *LNCS*, pages 286–303. Springer, 2003.

[7] M. M. Gorlick and C. F. Kesselman. Timing Prolog programs without clocks. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 426–435, San Francisco, Aug. - Sept. 1987. IEEE, Computer Society Press.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13(8):671–685, Aug. 1983.

[9] M. V. Hermenegildo, G. Puebla, K. Marriott, and P. J. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, Mar. 2000.

[10] A. B. Matos. A matrix model for the flow of control in Prolog programs with applications to profiling. *Software Practice and Experience*, 24(8):729–746, Aug. 1994.

[11] G. Puebla and M. V. Hermenegildo. Implementation of multiple specialization in logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 77–87, La Jolla, California, June 1995.

[12] G. Puebla and M. V. Hermenegildo. Optimized Algorithms for Incremental Analysis of Logic Programs. In R. Cousot and D. A. Schmidt, editors, *3rd International Symposium on Static Analysis*, volume 1145 of *LNCS*, pages 270–284, Aachen, Germany, Sept. 1996. Springer Verlag.

[13] H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 114–124, San Francisco, Aug. - Sept. 1987. IEEE, Computer Society Press.

[14] D. H. D. Warren. An Abstract PROLOG Instruction Set. Technical Report 309, Artificial Intelligence Center, Computer Science and Technology Division, SRI International, Oct. 1983.

[15] W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2-3):259–290, July 1992.

Improving the Compilation of Prolog to C Using Type and Determinism Information: Preliminary Results*

J. Morales† jfran@clip.dia.fi.upm.es

M. Carro† mcarro@fi.upm.es

M. Hermenegildo† ‡ herme@fi.upm.es

Abstract

We describe the current status of and provide preliminary performance results for a compiler of Prolog to C. The compiler is novel in that it is designed to accept different kinds of high-level information (typically obtained via an analysis of the initial Prolog program and expressed in a standardized language of assertions) and use this information to optimize the resulting C code, which is then further processed by an off-the-shelf C compiler. The basic translation process used essentially mimics an unfolding of a C-coded bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. Optimizations are then applied to this unfolded program. This is facilitated by a more flexible design of the bytecode instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: ancillary pieces of C code, data definitions, memory management routines and areas, etc., as well as mixing bytecode emulated code with natively compiled code in a relatively straightforward way. We report on the performance of programs compiled by the current version of the system, both with and without analysis information.

1 Introduction

Several techniques for implementing Prolog have been devised since the original interpreter developed by Colmerauer and Roussel [5], many of them aimed at achieving more speed. An excellent survey of a significant part of this work can be found in [26]. The following is a rough classification of implementation techniques for Prolog (which is, in fact, extensible to many other languages):

- Interpreters (such as C-Prolog [16] and others), where a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter.
- Compilers to *bytecode* and their interpreters (often called emulators). The compiler produces relatively low level code in a special-purpose language. An interpreter of such low-level code is still required. Most emulators are currently based on the Warren Abstract Machine (WAM) [28, 1], but other proposals exist [24, 13].
- Compilers to a lower-level language, typically ("native") machine code. In this case the output requires little or no additional support to be executed. One solution is for the compiler to generate machine code directly. Examples of this are the Aquarius system [27], versions of the SICStus Prolog [22] compiler for some architectures, the BIM-Prolog compiler [14], and the Gnu Prolog compiler [7]. Another alternative is to generate code in a (lower-level) language, such as, e.g., C-- [12] or C, for which a machine code compiler is readily available; the latter is the approach taken by `wamcc` [4].

*This work is partially supported by Spanish MCYT Project TIC 2002-0055 *CUBICO*, and EU IST Projects 2001-34717 *Amos* and IST-2001-38059 *ASAP*, and by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. J. Morales is also supported by an MCYT fellowship co-financed by the European Social Fund.

†Technical University of Madrid, Boadilla del Monte, E-28660, Spain.

‡Departments of Computer Science and Electrical and Computer Engineering, U. of New Mexico (UNM).

Each solution has its advantages and disadvantages:

Executable performance vs. executable size and compilation speed: Compilation to lower-level code can achieve faster programs by eliminating interpretation overhead and performing lower-level optimizations. In general, performing as much work as possible at compile time in order to avoid run-time overhead brings faster execution speed at the expense of using more resources during the compilation phase and possibly producing larger executables. In general, compilers are much more complex and take longer to preprocess programs for execution than interpreters. This difference gets larger as more sophisticated forms of code analysis are performed as part of the compilation process. This can impact development time, although sophisticated analyses can be turned off during development and applied when only generating production code. Interpreters in turn have potentially smaller load/compilation times and are often a good solution due to their simplicity when speed is not a priority. Emulators occupy an intermediate point in complexity and cost. Highly optimized emulators [19, 20, 6, 22, 2] offer very good performance and reduced program size (since single bytecode instructions correspond to several machine code instructions), which may be a crucial issue for very large programs and symbolic data sets.

Portability: Interpreters offer portability in a straightforward way since executing the same Prolog code in different architectures boils down (in principle) to simply recompiling the interpreter. Emulators usually retain the portability of interpreters, since only the emulator has to be recompiled for every target architecture (bytecode is usually architecture-independent), unless of course they are written in machine code.[1] Compilers to native code require architecture-dependent back-ends, i.e., a new translation into machine code has to be developed for each architecture. This typically makes porting and maintaining these compilers a non-trivial task. The task of developing these back-ends can be simplified by using an intermediate RTL-level code [7], although still different translations of this code are needed for different architectures.

Opportunities for optimizations: Code optimization can applied at all levels: to the Prolog level itself [18, 29], to WAM code [8], to lower-level code [15], and/or to native code [27, 23]. At a higher language level it is typically possible to perform more global and structural optimizations, which are then implicitly carried over onto lower levels. On the other hand, additional, lower-level optimizations can be introduced as we approach the native code level. These optimizations require exposing a level of detail in the operations that is not normally visible at higher levels. One of the most important motivations for compiling to machine code is precisely to be able to perform these low-level optimizations. In fact, recent performance evaluations show that well-tuned emulator-based Prolog systems can beat, at least in some cases, Prolog compilers which generate machine code directly but do not perform extensive optimization [7]. The approach of translating to a low-level language such as C is interesting because it makes portability straightforward, as C compilers exist for most architectures, and, on the other hand, C is low-level enough that it allows expressing in it a large class of low-level optimizations which will make into the final executable code in a form known beforehand, and which go beyond what can be expressed solely by means of Prolog-to-Prolog transformations.

Given all the considerations above, it is safe to say that different approaches are useful in different situations and perhaps even for different parts of the same program. In particular, the emulator approach with its competitive compilation times, program size, and performance, can be very useful during development, and in any case for non-performance bound portions of large symbolic data sets and programs. On the other hand, in order to generate the highest performance code it seems appropriate to perform optimizations at all levels and to eventually translate to machine code so that even the lowest-level optimizations can be performed, at least for parts of the program. The selection of a low-level language such as C as an intermediate target can offer a good compromise between opportunity for optimization and portability for native code.

In our compiler we have taken precisely such an approach: we use translation to C during which we apply several optimizations, making use of high-level information. Our starting point is the standard version of the Ciao Prolog system [2]. This is essentially an emulator-based system of quite competitive performance. Its abstract machine is an evolution of the &-Prolog abstract machine [10], itself a separate evolution branch from early versions (0.5–0.7) of the SICStus abstract machine. We have developed a compiler from Prolog to native code, via an intermediate translation to C, where the translation scheme adopts the same scheme for memory areas, data tagging, etc. as the emulator. This facilitates mixing emulated and native code (as done also by SICStus) and also has the important practical advantage that many complex and already

[1] This is the case for the Quintus emulator although it is coded in a generic RTL language ("PROGOL") to simplify ports.

existing fragments of C code present in the components of the emulator (such as builtins, low-level file and stream management, memory management and garbage collection routines, etc.) can be reused by the new compiler. This is important because our intention is to develop not a prototype but a full compiler that can be put into everyday use and it would be an unrealistic amount of work to develop all those parts again. Also, compilation to C allows us to translate Prolog modules into C files that can be compiled as source files in multi-language applications.

As mentioned before, the selection of C as target low-level language allows performing a large class of low-level optimizations while easing portability. A practical advantage in this sense is the availability of C compilers for most architectures, such as `gcc`, which generate very efficient executable code. The difference with other systems which compile to C comes from the fact that the translation that we propose includes a scheme for optimizing the resulting code using higher-level information available at compile-time regarding determinacy, types, instantiation modes, etc. of the source program. The objective is better run-time performance, including a reduction of the size of executables. We also strive to preserve portability and maintainability by avoiding the use of non-standard C code as much as possible.

This line of reasoning lead us also not to adopt other approaches such as compiling to C--. The goal of C-- is to achieve portable high performance without relinquishing control over low-level details. However, the associated tools do not seem to be presently mature enough as to be used for a compiler in production status within a near future, and not even to be used as base for a research protoype in their present stage. Future portability will also depend on the existence of back-ends for a range of architectures. We, however, are quite confident that the backend which now generates C code could be adapted to generate C-- (or other low-level languages) without too many problems.

The high-level information is expressed by means of a powerful and well-defined assertion language [17], and inferred by automatic global analysis tools which code the results of analysis as assertions or simply provided by the user. In our system we take advantage of the availability of relatively mature tools for this purpose within the Ciao system, and, in particular the preprocessor, CiaoPP [11].

Our approach is thus different from, for example, `wamcc` (the forerunner of the current Gnu Prolog), which also generated C, but did not use extensive analysis information (and it used low-level, clever tricks which in practice tied it to a particular C compiler, `gcc`). Aquarius [27] (and [23]) used analysis information at several compilation stages, but they generated directly machine code, and it has proved difficult to port and maintain these systems. Also, program analysis technology was not as mature at the time as it is now. Notwithstanding, they were landmark contributions that proved the power of using global information in a Prolog compiler.

A drawback of putting more burden on the compiler is that compile times grow, and compiler complexity increases, specially in the global analysis phase. While this can turn out to be a problem in extreme cases, incremental analysis in combination with a suitable module system [3] can result in very reasonable analysis times in practice. Moreover, global analysis (or even the compilation to C) are not mandatory in our proposal and can be reserved for the phase of generating the final, "production" executable. We expect that, as the system matures, the Prolog-to-C compiler itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

2 The Basic Compilation Scheme

We now present the basic compilation strategy. The optimizing compilation using global program information will be described in Section 3.

The compilation process starts with a preprocessing phase which canonizes clauses (removing aliasing and structure unification from the head, also known as "normalization"), and expands disjunctions, negations and if-then-else constructs. It also unfolds calls to `is/2` when possible into calls to simpler arithmetic predicates, replaces the cut by calls to the lower-level predicates `metachoice/1` (which stores in its argument the address of the current choicepoint) and `metacut/1` (which performs a cut to the choicepoint whose address is passed in its argument), and performs a simple, local analysis which gathers information about the type and freeness state of variables.[2] Having this analysis in the compiler (in addition to the analyses

[2]In general, the types used throughout the paper are *instantiation types*, i.e., they have mode information built in (see [17] for

put_variable(I,J)	⟨uninit,I⟩ = ⟨uninit,J⟩
put_value(I,J)	⟨init,I⟩ = ⟨uninit,J⟩
get_variable(I,J)	⟨uninit,I⟩ = ⟨init,J⟩
get_value(I,J)	⟨init,I⟩ = ⟨init,J⟩
unify_variable(I[, J])	`if` (initialized(J)) `then` ⟨uninit,I⟩ = ⟨init,J⟩ `else` ⟨uninit,I⟩ = ⟨uninit,J⟩
unify_value(I[, J])	`if` (initialized(J)) `then` ⟨init,I⟩ = ⟨init,J⟩ `else` ⟨init,I⟩ = ⟨uninit,J⟩

Table 1: Representation of some WAM unification instructions with types.

performed by the preprocessor) allows improving the code even in the case that no external information is available from previous stages or the user. The following steps of the compiler include the translation from this normalized version of Prolog to WAM-based instructions (at this point the same ones used by the Ciao emulator), and then splitting these WAM instructions into an intermediate low level code and performing the final translation to C.

Typing WAM Instructions: WAM instructions dealing with data are handled internally using an enriched representation which encodes the possible instantiation state of their arguments.

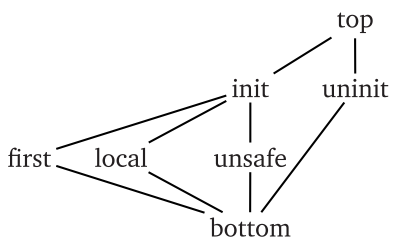


Figure 1: Lattice of WAM types.

This allows using original type information, and also generating and propagating lower-level information regarding the type (i.e., from the point of view of the tags of the abstract machine) and instantiation/initialization state of the variables (which is not seen at a higher level). Each unification instruction is represented as ⟨TypeX, X⟩ = ⟨TypeY, Y⟩, where *TypeX* and *TypeY* refer to the classification of WAM-level types (see Figure 1), and *X* and *Y* refer to variables, which may be later stored as WAM X or Y registers or directly passed on as C function arguments.

Table 1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers *x(I)*, the stack registers *y(I)* and the register for structure arguments *n(I)*. The last one can be seen as the second argument which is implicit in the *unify_** WAM instructions. A number of other special-purpose registers (`ok`, `temp`, ...) are available, and used, for example, to hold intermediate results from expression evaluation and to record whether a builtin failed. **_constant*, **_nil*, **_list* and **_structure* WAM instructions are represented similarly. Only register variables *x(·)* are created in an uninitialized state, and they are initialized on demand (in particular, when calling another predicate which may overwrite the registers, and in the points where garbage collection can start). Stack and structure (heap) variables are created initialized.

One of the advantages of this representation is that it is more uniform than the traditional WAM instructions. In particular, as more information is known about the variables, the associated (low level) types can be refined and more specific code generated. Using a richer lattice and initial information (Section 3), a more descriptive intermediate code is generated and used in the back-end.

Generation of the Intermediate Low Level Language: WAM instructions are then split into simpler ones, which are more suitable for optimizations. This also allows simplifying the generation of the final C code (and probably also the generation of code in other languages of similar abstraction level). The degree of complexity of the low-level code is similar to the one proposed in the BAM [25]. Table 2 summarizes the instructions. The *Type* argument which appears in several of them is intended to reflect the type of the instruction arguments: for example, in the instruction *bind*, *Type* used to specify if the arguments contain a

a more complete discussion of this issue). *Freeness of variables* distinguishes between free variables and the *top* type, "term", which includes any term.

Choice, stack and heap management instructions	
no_choice	Mark that there is no alternative
first_choice(Arity, Alt)	Create a choicepoint
middle_choice(Arity, Alt)	Change the alternative
last_choice(Arity)	Remove the alternative
complete_choice(Arity)	Complete the choice point
cut_choice(Chp)	Cut to a given choice point
push_frame	Allocate a frame on top of the stack
complete_frame(FrameSize)	Complete the stack frame
modify_frame(NewSize)	Change the size of the frame
pop_frame	Deallocate the last frame
recover_frame	Recover after returning from a call
ensure_heap(CS, Amount, Arity)	Ensure that enough heap is allocated. (CS indicates completion status of the choice point)
Data	
load(X, Type)	Load *X* with a term
trail_if_conditional(A)	Trail if *A* is a conditional variable
bind(TypeX, X, TypeY, Y)	Bind *X* and *Y*
read(Type, X)	Begin read of the structure arguments of *X*
deref(X, Y)	Dereference *X* into *Y*
move(X, Y)	Copy *X* to *Y*
globalize_if_unsafe(X, Y)	Copy *X* to *Y* ensuring safety
globalize_to_arg(X, Y)	Copy *X* to argument register *Y* ensuring safety
call(CallerImp, CalledImp, Pred, In, Out, FailCont)	Call a builtin or a user predicate. *CallerImp* and *CalleeImp* mark how caller and callee are compiled.
Control	
ijump(X)	Jump to the address stored in *X*
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
switch_on_type(X, Var, Str, List, Cons)	Jump to the label that matches the type of *X*
switch_on_functor(X, Table, Else)	
switch_on_cons(X, Table, Else)	
Conditions	
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if *X* matches *Type*
equal(X, Y)	True if *X* and *Y* are equal
erroneous(X)	True if *X* has an erroneous value

Table 2: Control and data instructions.

variable (and, if this is known, whether it lives in the heap, in the stack, etc.) or not. For the unification of structures, the use of write and read modes is avoided using a two-stream scheme (see [26] for an explanation and references) which is encoded using with the unification instructions in Table 1 and later translated into the required series of assignments. This scheme requires explicit control instructions, hence the existence of jump instructions (*jump*, *cjump*, and *ijump*). Jumps are performed to labels, marked as *global* (when they have to be stored in global data structures, such as the next alternative in a choicepoint) or *local*. For efficiency in indexing, the WAM instructions *switch_on_term*, *switch_on_cons* and *switch_on_functor* are also included, although the C back-end does not exploit them fully at the moment, resorting to a linear search in some cases. Failing is done by jumping to the special label *fail*. Builtins return an exit state in one argument, which is used to decide whether to backtrack or not. Determinism information, when available, is passed on through this stage and used when compiling with optimizations (see Section 3).

Compilation to C: This stage in turn produces the output C code. This C code conceptually corresponds to an unfolding of the initial bytecode emulator loop with respect to the particular sequence(s) of bytecode corresponding to the program. In the points where the emulated program counter changes continuations are passed using pointers to functions. Each block of bytecode (i.e., each sequence beginning in a label and ending in an instruction involving a possible jump) is translated to an individual C function. The state of the

```
while (code != NULL)
   code = ((Continuation (*)(State *))code)(state);
```

```
Continuation foo(State *state) {
   ...
   state->cont = &foo_cont;
   return &foo2;
}
```

```
Continuation foo_cont(State *state) {
  ...
  return state->cont;
}
```

Figure 2: The C execution loop and blocks scheme.

abstract machine is the input argument to the function and the next continuation is the output argument. This approach avoids building functions that are too large and would create problems for the C compiler. Figure 2 shows schematic versions of the execution loop and of the functions that code blocks are compiled into. The translation also incorporates an optimization which reduces the function calling overhead: `goto` statements are used for jumps to local labels which are located in the same code block.

This scheme does not require using machine-dependent options of the C compiler or extensions to the ANSI C language (although machine-dependent optimizations can of course be given to the C compiler). Other systems, as [20] or [21], take advantage of machine-dependent and non-portable constructs to increase performance. However, one of the goals of our system –to study the impact of optimizations based on available information on the program– can be achieved with the proposed compilation scheme, and, as mentioned before, we give portability and code cleanliness a high priority. The possibility of producing more efficient but non-portable code can always be added at a later stage.

An Example — the `fact/2` Predicate: We will illustrate briefly the different stages of compilation using the well-known factorial program (Figure 3). We have chosen it due to its simplicity, even if the performance gain is not very high in this case. The code after the first canonizing and rewriting stage is shown in Figure 4. The WAM code corresponding to the recursive clause is listed in the leftmost column of Table 3, and the internal representation of this code appears in the same table, in the middle column. Note how variables are annotated using information which can be deduced from local inspection of the clause.

This WAM-like representation is translated to the low-level code shown in Figure 5 (ignore, for the moment, the shadowed and framed regions; they will be further discussed in Section 3). This code, which is quite low-level now, is what is finally translated to C.

For reference, executing `fact(100, N)` 20000 times took 0.65 seconds running emulated bytecode, and 0.63 seconds running the code compiled to C (a speedup of 1.03). This was all done without using any external, global type information. In the next section we will see how this performance can be improved with the use of type information.

```
fact(0, 1).
fact(X, Y) :-
    X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.
```

Figure 3: Factorial, initial code.

```
fact(A, B) :-
    0 = A,
    1 = B.
```

```
fact(A, B) :-
    A > 0,
    builtin__sub1_1(A, C),
    fact(C, D),
    builtin__times_2(A, D, B).
```

Figure 4: Factorial, after preprocessing.

3 Improving Code Generation

As mentioned in Section 1, our objective is to improve the code generation process using information regarding global properties of predicates which is coded as assertions [17] –a few such assertions can be seen in the example of Section 3. In the current version of the compiler optimization is performed using information on instantiation types (i.e., moded types) as well as number of solutions (determinacy).

WAM code	Without Types	With Types
put_constant(0,2)	0 = ⟨uninit,x(2)⟩	0 = ⟨uninit,x(2)⟩
builtin_2(37,0,2)	⟨init,x(0)⟩ > ⟨int(0),x(2)⟩	⟨int,x(0)⟩ > ⟨int(0),x(2)⟩
allocate	builtin__push_frame	builtin__push_frame
get_y_variable(0,1)	⟨uninit,y(0)⟩ = ⟨init,x(1)⟩	⟨uninit,y(0)⟩ = ⟨var,x(1)⟩
get_y_variable(2,0)	⟨uninit,y(2)⟩ = ⟨init,x(0)⟩	⟨uninit,y(2)⟩ = ⟨int,x(0)⟩
init([1])	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩
true(3)	builtin__complete_frame(3)	builtin__complete_frame(3)
function_1(2,0,0)	builtin__sub1_1(⟨init,x(0)⟩, ⟨uninit,x(0)⟩)	builtin__sub1_1(⟨int,x(0)⟩, ⟨uninit,x(0)⟩)
put_y_value(1,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩
call(fac/2,3)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨init,x(1)⟩)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨var,x(1)⟩)
put_y_value(2,0)	⟨init,y(2)⟩ = ⟨uninit,x(0)⟩	⟨int,y(2)⟩ = ⟨uninit,x(0)⟩
put_y_value(2,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨number,y(1)⟩ = ⟨uninit,x(1)⟩
function_2(9,0,0,1)	builtin__times_2(⟨init,x(0)⟩, ⟨init,x(1)⟩,⟨uninit,x(0)⟩)	builtin__times_2(⟨int,x(0)⟩, ⟨number,x(1)⟩, ⟨uninit,x(0)⟩)
get_y_value(0,0)	⟨init,y(0)⟩ = ⟨init,x(0)⟩	⟨var,y(0)⟩ = ⟨init,x(0)⟩
deallocate	builtin__pop_frame	builtin__pop_frame
execute(true/0)	builtin__proceed	builtin__proceed

Table 3: WAM code and internal representation without and with external types information. Underlined instruction changed due to additional information.

The generation of low-level code using additional type information makes use of an extended type lattice obtained by replacing the *init* element in the lattice in Figure 1 with the type domain in Figure 6. This information enriches the *Type* parameter of the low-level code. Additionally, as mentioned before, any information about the determinacy / number of solutions of each call is carried over into this stage and used in it to optimize the generated C code.

In general, information about the types of variables and determinism of predicates allows avoiding introducing unnecessary tests during the compilation to low level code. The standard WAM compilation performs

```
global(fact/2):
  first_choice(2,V1)
  ensure_heap(incompleted_choice,callpad,2)
  deref(x(0),x(0))
  cjump(not(test(var,x(0))),local(V3))
  load(temp2,int(0))
  bind(var,x(0),nonvar,temp2)
  jump(local(V4))
local(V3):
  cjump(not(test(int(0),x(0))),fail)
local(V4):
  deref(x(1),x(1))
  cjump(not(test(var,x(1))),local(V5))
  load(temp2,int(1))
  jump(local(V6))
local(V5):
  cjump(not(test(int(1),x(1))),fail)
local(V6):
  complete_choice(2)
  ijump(continuation)
global(V1):
  last_choice(2)
  load(x(2),int(0))
  builtin(numgt_2,[x(0),x(2)],ok)
  cjump(not(ok),fail)
  push_frame
  move(x(1),y(0))
  move(x(0),y(2))
  load(y(1),var(stack))
  complete_frame(3)
  function(sub1_1,[x(0)],x(0),0,1)
  cjump(erroneous(x(0)),fail)
  move(y(1),x(1))
  modify_frame(3)
  load(continuation,global(V0))
  jump(global(fact/2))
global(V0):
  recover_frame
  move(y(2),x(0))
  move(y(1),x(1))
  function(times_2,[x(0),x(1)],x(0),0,2)
  cjump(erroneous(x(0)),fail)
  deref(y(0),temp)
  deref(x(0),x(0))
  builtin(unify,[temp,x(0)],ok)
  cjump(not(ok),fail)
  pop_frame
  ijump(continuation)
```

Figure 5: Low level code for the fact/2 example (see also Section 3).

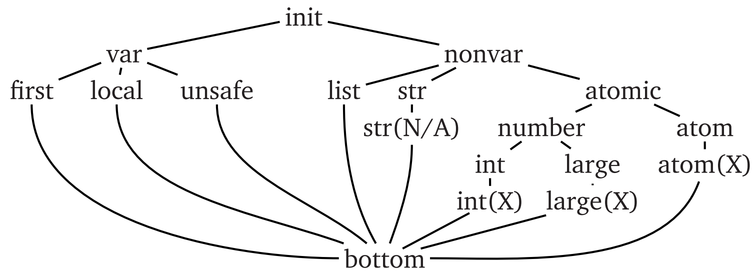


Figure 6: Extended *init* subdomain.

also some optimizations (e.g., classification of variables and indexing on the first argument), but it is based on a per-clause (per-predicate, in the case of indexing) based analysis, and in general it does not propagate, e.g., information deduced from arithmetical builtins. A number of further optimizations can be done by using richer type, mode, and determinism information:

Unify instructions: A call to the general *unify* builtin is replaced by the more specialized *bind* instruction if one or both arguments are known to store variables. When arguments are known to be constants, a simple comparison instruction is emitted instead.

Two-Stream Unification: The unification of a register with a structure or constant requires some tests for determining the unification mode (read or write). Also, in read mode, an additional test is required to compare the register value with the constant or the structure functor. These tests can often be reduced to true or false at compile-time if enough information is known about the variable.

Index Tree Generation: Type information is also used to optimize the generation of index trees, which are used as part of the clause selection process. An index tree is generated by selecting some literals from the beginning of the clause, mostly builtins and unifications, which give some amount of type/mode information. This is used to construct a decision tree on the types of the first argument.[3] When type information is available, the indexing tree can be optimized by removing some of the tests in the nodes.

Avoiding Unnecessary Variable Safety Tests: Another optimization performed in the low level code using type information is the replacement of globalizing instructions for unsafe variables by explicit dereferences. When the type of a variable is nonvar, its globalization is equivalent to a dereference, which is faster.

Uninitialized Output Arguments: When possible, letting the called predicate fill in the contents of output arguments in pre-established registers avoids allocation, initialization and binding of free variables, which is slower.

Selecting Optimized Predicate Versions: Calls to predicates can also be optimized in the presence of type information. Specialized versions (in the sense of low level optimizations) can exist and be selected using the call patterns deduced from the type information. The current implementation does not support automatic versions of user predicates (since this is done automatically by the preprocessor[18]), but it does optimize natively internal *builtin* predicates written in C (such as, e.g., arithmetic builtins) which results in relevant speedups in many cases.

[3]This can of course be extended to other arguments.

Determinism: These optimizations are based on two types of analysis. The first one uses information regarding the number of solutions for a predicate call to deduce, for each such call, if there is a known and fixed fail continuation. Then, instructions to create choicepoints and to restore previous choicepoint states are inserted. The resulting code is then re-analyzed to remove these instructions when possible or to replace them by simpler ones (e.g., to restore a choice point state without untrailing, if it is known at compile time that the execution will not trail any value since the choice point was created). The latter can take advantage of additional information regarding register, heap, and trail usage of each predicate.[4] In addition, the C back-end can generate different argument passing schemes based on determinism information: predicates with zero or one solution can be translated to a function returning a boolean, and predicates with exactly one solution to a function returning `void`.

An Example — the `fact/2` Predicate with program information: Let us assume that it is known that `fact/2` (Figure 3) is always called with its first argument instantiated to a small integer (an integer which fits into a tagged word of the internal representation) and its second argument is a free variable. This information can be written in the assertion language as follows:[5]

```
:- true pred fact(X, Y) :  int * var => int * number.
```

which reflects the types and modes of the calls and successes of the predicate. The propagation of that information through the canonized predicate gives the annotated program shown in Figure 7.

```
fact(A, B) :-
    true(int(A)),
    0 = A,
    true(var(B)),
    1 = B.
```

```
fact(A, B) :-
    true(int(A)),
    A > 0,
    true(int(A)), true(var(C)),
    builtin__sub1_1(A, C),
    true(any(C)), true(var(D)),
    fact(C, D),
    true(int(A)), true(number(D)),
    true(var(B)),
    builtin__times_2(A, D, B).
```

Figure 7: Annotated factorial (using type information).

The WAM code generated for this example is shown in the rightmost column of Table 3. Underlined instructions were made more specific due to the initial information — note, however, that the representation is homogeneous with respect to the "no information" case. The impact of type information in the generation of low-level code can be seen in Figure 5. Instructions in the shaded regions are **removed** when type information is available, and the (arithmetic) builtins enclosed in rectangles are replaced by calls to versions specialized to work with small integers and which do not perform type/mode testing. The optimized `fact/2` program took 0.54 seconds with the same call as in Section 2: a 20% speedup with respect to the bytecode version and a 16% speedup over the compilation to C without type information.

4 Performance Measurements

We have evaluated the performance behavior of the executables generated with our compiler using translation to C with respect to that of the emulated bytecode on a set of standard benchmarks. The benchmarks are not real-life programs, and some of them have been executed up to 10.000 times in order to obtain reasonable and stable execution times. All the measurements have been made in a Pentium 4 Xeon @ 2.0GHz with 1Gb of RAM, running Linux with a 2.4 kernel and using `gcc` 3.2 as C compiler. A short description of the benchmarks follows:

[4] This is currently known only for internal predicates written in C, and which are available by default in the system, but the scheme is general and can be extended to Prolog predicates.

[5] The `true` prefix implies that this information is to be *used*, rather than to be *checked* by the compiler.

Program	Bytecode (Std. Ciao)	Non opt. C	Opt1. C	Opt2. C
`queens11 (1)`	691	391 (1.76)	208 (3.32)	166 (4.16)
`crypt (1000)`	1525	976 (1.56)	598 (2.55)	597 (2.55)
`primes (10000)`	896	697 (1.28)	403 (2.22)	402 (2.22)
`tak (1000)`	9836	5625 (1.74)	5285 (1.86)	771 (12.75)
`deriv (10000)`	125	83 (1.50)	82 (1.52)	72 (1.74)
`poly (100)`	439	251 (1.74)	199 (2.20)	177 (2.48)
`qsort (10000)`	521	319 (1.63)	378 (1.37)	259 (2.01)
`exp (10)`	494	508 (0.97)	469 (1.05)	459 (1.07)
`fib (1000)`	263	245 (1.07)	234 (1.12)	250 (1.05)
`knights (1)`	621	441 (1.40)	390 (1.59)	356 (1.74)
Average Speedup		(1.47)	(1.88)	(3.18)

Table 4: Bytecode emulation vs. unoptimized, optimized (types), and optimized (types and determinism) compilation to C.

Program	GProlog	WAMCC	SICStus	SWI	Yap	Mercury	$\frac{\text{Opt2. C}}{\text{Mercury}}$
`queens11 (1)`	809	378	572	5869	362	106	1.57
`crypt (1000)`	1258	966	1517	8740	1252	160	3.73
`primes (10000)`	1102	730	797	7259	1233	336	1.20
`tak (1000)`	11955	7362	6869	74750	8135	482	1.60
`deriv (10000)`	108	126	121	339	100	72	1.00
`poly (100)`	440	448	420	1999	424	84	2.11
`qsort (10000)`	618	522	523	2619	354	129	2.01
`exp (10)`	—	—	415	—	340	—	—
`fib (1000)`	—	—	285	—	454	—	—
`knights (1)`	911	545	631	2800	596	135	2.63
						Average	1.98

Table 5: Speed of other Prolog systems and Mercury

crypt: Cryptoarithmetic puzzle involving multiplication.
primes: Sieve of Erathostenes (with N = 98).
tak: Computation of the Takeuchi function with arguments `tak(18, 12, 6, X)`.
deriv: Symbolic derivation of polynomials.
poly: Symbolically raise `1+x+y+z` to the 10^{th} power.
qsort: QuickSort of a list of 50 elements.
exp: Computation of 13^{7111} using both a linear- and a logarithmic-time algorithm.
fib: Computation of F_{1000} using a simply recursive predicate.
knight: Chess knight tour (visit only once all the board cells) in a 5×5 board.

A summary of the results appears in Table 4. The number between parentheses in first column is the number of iterations of each benchmark (used to obtain an execution long enough). The second column contains the execution times of programs compiled to bytecode (this represents the speed of the standard Ciao bytecode emulator). The third column corresponds to programs compiled to C without compile-time information (which corresponds, basically, to mimicking the bytecode execution). The fourth and fifth columns correspond, respectively, to the execution times when compiling to C using type information and type+determinism information to optimize the resulting code. The numbers between parentheses are the speedups relative to the bytecode version. All times are in milliseconds.

In order to know how these numbers compare with the performance of other Prolog systems, Table 5 shows the execution times (also in milliseconds) for the same benchmarks in four well-known Prolog compilers: GNU Prolog 1.2.16, wamcc 2.23, SICStus 3.8.6, SWI-Prolog 5.2.7, and Yap 4.5.0. The aim here is not really to compare directly with them, because a different technology is being used (compilation to C and use of external information which they cannot take advantage of), but rather to establish that our baseline, the speed of the bytecode system (Ciao), is similar (and quite close, in particular, to that of SICStus and Yap). Thus, in principle, comparable optimizations could be made in these systems. YAP was itself compiled with multi-precision arithmetic, which makes its execution a little bit slower than without it (just some milliseconds, not enough as to make a significant difference). The cells marked with "—" correspond to cases where the benchmark could not be executed (in GNU Prolog, wamcc, and SWI, due to lack of multi-precision arithmetic).

We also include the performance results for the compiler for the Mercury language [21] (version 0.11.0). Strictly speaking the Mercury compiler is not a Prolog compiler, since the source language is substantially different from Prolog. On the other hand the Mercury language does have enough similarities to be relevant and its performance is interesting as an upper reference line given that the language was designed precisely to allow the compiler, which directly generates machine code, to achieve very high performance by using extensive low-level optimization compilation techniques. Also, the language design requires that the programs necessarily contain as part of the source the necessary information to perform these optimizations.

Going back to Table 4, while some performance gains are obtained in the *naive* translation to C, such gains are not very significant, and there is even one program which shows a slowdown. We have tracked this down to be due to several factors:

- The simple compilation scheme generates C code that is as clean and portable as possible, avoiding tricks which would speed up the programs. The execution profile is also very near to what the emulator would do.
- The C execution loop (Figure 2) is slightly more costly (by a few assembler instructions) than the fetch/switch loop of the emulator. We have identified this as a cause for the poor speedup of programs where recursive calls dominate the execution time. We want, of course, to improve this point in the future.
- The increment in size of the program (see Table 6) may also cause more cache misses. We also want to investigate this point in more detail.

As expected, the performance obtained when using compile-time information is much better. The best speedups are obtained in benchmarks using arithmetic builtins, for which the compiler produces optimized versions where several groundness and type checks have been removed. This is, for example, the case of `queens`, in which it is known that all the numbers involved are small integers (i.e., there is no need for infinite precision arithmetic). Besides avoiding checks, the functions which implement the arithmetic operations for small integers are simple enough as to be inlined by the C compiler. This is an example of an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it. When determinism information is used, the execution is often (but not always) improved. The Takeuchi function (`tak`) is an extreme case, where determinism information saves choicepoint generation and execution time. While the performance obtained is still far (a factor of 2 on average) from that of Mercury, the results are encouraging given that we are dealing with a more complex source language (which preserves fully unification, logical variables, etc.), we are using a portable approach (compilation to standard C), and we have not applied yet all possible optimizations.

Table 6 compares object size of the bytecode and the different schemes of compilation to C. Unit is bytes. As mentioned in Section 1, due to the different granularity of instructions, larger object files and executables are expected when compiling to C. The ratio depends heavily on the program and the optimizations applied. Size increase can be as large as $15\times$ when translating to C without optimizations, and the average case sits around a 7-fold increase. This is also partially due to the indexing mechanism, which repeats some code. We plan to improve this in the future. It must also be pointed out that executing the bytecode requires always the presence of the bytecode emulator, which is around 300Kb (depending on the architecture and the optimizations applied) and which should be added to the figures in Table 6 for the emulator. On the other hand this can be shared among different executables as a library. The executables obtained through C do not need the emulation loop, and only the code for the GC routines and the definition of predicates internally written in C has to be linked at runtime.

The size of the object code produced by `wamcc` is roughly comparable to that generated by our compiler, although `wamcc` produces smaller object code files. However the final executable / process size depends also

Program	Bytecode	Non opt. C	Opt1. C	Opt2. C
`queens11`	7167	36096 (5.03)	29428 (4.10)	42824 (5.97)
`crypt`	12205	186700 (15.30)	107384 (8.80)	161256 (13.21)
`primes`	6428	50628 (7.87)	19336 (3.00)	31208 (4.85)
`tak`	5445	18928 (3.47)	18700 (3.43)	25476 (4.67)
`deriv`	9606	46900 (4.88)	46644 (4.85)	97888 (10.19)
`poly`	13541	163236 (12.05)	112704 (8.32)	344604 (25.44)
`qsort`	6982	90796 (13.00)	67060 (9.60)	76560 (10.96)
`exp`	6463	28668 (4.43)	28284 (4.37)	25560 (3.95)
`fib`	5281	15004 (2.84)	14824 (2.80)	18016 (3.41)
`knights`	7811	39496 (5.05)	39016 (4.99)	39260 (5.03)
Average Increase		(7.39)	(5.43)	(8.77)

Table 6: Compared size of object files (bytecode vs. C).

on which libraries are linked statically and/or dynamically. The Mercury system is somewhat incomparable in this regard: it certainly produces relatively small component files but then relatively large final executables (over 1.5 MByte).

The size, in general, decreases when using type information, as many dynamic type tests are removed. The average size is now around five times the bytecode size. Adding determinism information increases the code size because of the additional inlining performed by the C compiler and the more complex parameter passing code. The options passed on to the C compiler were the same in all the programs, and the decision of whether to inline or not was left to it. Some experiments showed that asking the C compiler to do more aggressive inlining did not help to achieve better speedups.

It is interesting that some of the optimizations used in the compilation to C would not give comparable results when applied directly to a bytecode emulator. A version of the bytecode emulator hand-coded to work only with small integers (which can be boxed into a tagged word) performed worse than that obtained doing the same with compilation to C. That suggests that when the overhead of calling builtins is reduced, as is the case in the compilation to C, some optimizations which only produce minor improvements for emulated systems acquire greater importance.

5 Conclusions and Future Work

We have reported on the scheme and performance of a Prolog-to-C compiler which uses type analysis and determinacy information to improve the final code by removing type and mode checks and by making calls to specialized versions of some builtins. We have also provided preliminary performance results for this compiler. Our compiler is still in a preliminary stage, but already shows promising results.

The compilation uses internally a simplified and more homogeneous representation for WAM code, which is then translated to a lower-level intermediate code. This step uses the type and determinacy information available at compile time. This code is finally translated into C by the compiler back-end. The intermediate code, as in other similar compilers, makes the final translation step easier and will make it easier to develop new back-ends for other target languages.

We have found using the same information to optimize a WAM bytecode emulator to be more difficult and to result in lower speedups, due to the greater granularity of the bytecode instructions (which aims at reducing the cost of fetching them). The same result has been reported elsewhere [26], although some recent work tries to improve WAM code by means of local analysis [9, 8].

We expect to be able to use information (e.g., determinacy) to improve also clause selection, as well as to generate a better indexing scheme at the C level by using hashing on constants, instead of the linear search performed now. Also, we want to study which other optimizations can be added to the generation of C code without breaking its portability, and how the intermediate representation can be used to generate code for other back-ends (for example, GCC RTL, CIL, Java bytecode, etc.).

References

[1] H. Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual (v1.8). The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002. System and on-line version of the manual available at `http://clip.dia.fi.upm.es/Software/Ciao/`.

[3] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[4] P. Codognet and D. Diaz. WAMCC: Compiling Prolog to C. In L. Sterling, editor, *International Conference on Logic Programming*, pages 317–331. MIT PRess, June 1995.

[5] A. Colmerauer. The Birth of Prolog. In *Second History of Programming Languages Conference*, ACM SIGPLAN Notices, pages 37–52, March 1993.

[6] B. Demoen and P.-L. Nguyen. So Many WAM Variations, So Little Time. In *Computational Logic 2000*, pages 1240–1254. Springer Verlag, July 2000.

[7] D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[8] M. Ferreira and L. Damas. Multiple Specialization of WAM Code. In *Practical Aspects of Declarative Languages*, number 1551 in LNCS. Springer, January 1999.

[9] M. Ferreira and L. Damas. Wam local analysis. In B. Demoen, editor, *Proceedings of CICLOPS 2002*, pages 13–25, Copenhagen, Denmark, June 2002. Department of Computer Science, Katholieke Universiteit Leuven.

[10] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[11] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[12] S. L. P. Jones, N. Ramsey, and F. Reig. C--: A Portable Assembly Language that Supports Garbage Collection. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28. Springer Verlag, September 1999.

[13] A. Krall and T. Berger. The $\mathrm{VAM}_{\mathrm{AI}}$ - an abstract machine for incremental global dataflow analysis of Prolog. In M. G. de la Banda, G. Janssens, and P. Stuckey, editors, *ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages*, pages 80–91, Tokyo, 1995. Science University of Tokyo.

[14] A. Mariën. *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*. PhD thesis, Katholieke Universiteit Leuven, September 1993.

[15] J. Mills. A high-performance low risc machine for logic programming. *Journal of Logic Programming (6)*, pages 179–212, 1989.

[16] F. Pereira. *C-Prolog User's Manual, Version 1.5*. University of Edinburgh, 1987.

[17] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[18] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[19] *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[20] V. Santos-Costa, L. Damas, R. Reis, and R. Azevedo. *The Yap Prolog User's Manual*, 2000. Available from `http://www.ncc.up.pt/~vsc/Yap`.

[21] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[22] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, Oct. 1999. Available from `http://www.sics.se/sicstus/`.

[23] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. In *1990 International Conference on Logic Programming*, pages 174–189. MIT Press, June 1990.

[24] A. Taylor. *High-Performance Prolog Implementation*. PhD thesis, Basser Department of Computer Science, Unversity of Sidney, June 1991.

[25] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, Univ. of California Berkeley, 1990. Report No. UCB/CSD 90/600.

[26] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.

[27] P. Van Roy and A. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[28] D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[29] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.

Incremental Copying Garbage Collection for WAM-based Prolog systems

Ruben Vandeginste Bart Demoen

Department of Computer Science,
Katholieke Universiteit Leuven, Belgium
{ruben,bmd}@cs.kuleuven.ac.be

Abstract

We present an incremental copying garbage collector for the heap in WAM-based Prolog systems. We describe a heap layout in which the heap is divided in a number of equal-sized blocks. Other changes to the standard WAM allow to garbage collect these blocks independently from each other. Independent collection of heap blocks is the basis of our incremental algorithm. Compared to other copying collectors for Prolog, this collector has several advantages. First of all, it reduces pause times by only collecting one heap block at a time. Second, our algorithm has in many cases a lower memory usage than standard semi-space copying collectors. Our algorithm is based on copying without marking (contrary to the more frequently used mark-copy algorithms in the context of Prolog); but while usually a copying collector needs a *to space* as big as the size of its *from space*, our collector needs a *to space* only the size of one heap block. Another benefit is that this algorithm also allows for a variety of garbage collection policies (including generational ones).

1 Introduction

We assume some basic knowledge about Prolog and its implementation. The WAM [13, 1] (Warren Abstract Machine) is a well-known virtual machine with a specialized instruction set for the execution of Prolog code. The WAM has proven to be a good basis for efficient Prolog implementations, and a lot of systems are based on it. We also assume basic knowledge about garbage collection in general; a good overview is given in [14, 11]. Some good references related to garbage collection for Prolog are [2, 3, 4, 9].

The WAM defines 3 different memory areas: a merged stack for environments and choicepoints, a trail, and a heap. Different memory management techniques are available to recover space in these areas. In this work we focus on memory management of the heap. The basic WAM already provides a mechanism to recover space allocated on the heap. Each time the system backtracks, it can deallocate all data allocated since the creation of the most recent choicepoint. We define a heap segment as the space on the heap, delimited by the heap pointers in 2 consecutive choicepoints. So, upon backtracking, the WAM can recover all space on the heap allocated for the most recent heap segment. This technique for recovering heap space is called instant reclaiming.

In practice however, instant reclaiming alone is not sufficient. Also, many Prolog programs are rather deterministic, in which case instant reclaiming is not effective. Because of this, Prolog systems need garbage collection for the heap. Early on, many systems used mark-slide garbage collection [2] because of the following properties. Mark-slide garbage collection preserves the cell order and as a consequence also preserves heap segments (which is important for instant reclaiming).

Also important is memory usage; besides mark and chain bits (2 extra bits per heap cell), no extra space is needed for garbage collection.

Lately copying garbage collection has become more popular. Contrary to mark-slide garbage collection, copying garbage collection does not preserve the cell order, nor the heap segments and consequently loses the ability to do instant reclaiming. A copying garbage collector, however, has better complexity than a mark-slide garbage collector and in most cases copying collectors outperform mark-slide collectors.

Still, there is an issue with semi-space copying collectors in Prolog: during garbage collection extra cells might be created and this can cause the *to space* to overflow. These collectors are considered *unsafe*. A detailed discussion of this problem can be found in [9]. Copying collectors can be made *safe* by adding a marking phase and most copying collectors in the context of Prolog are in fact mark-copy collectors [4]. Recent work in [9], however, has shown that *copying without marking* is still pretty *safe* and that a simple change in the copying algorithm (*optimistic copying*) can make the problem less severe. Moreover, eliminating the marking phase gives a considerable performance improvement. We use *optimistic copying* as the basis for our incremental algorithm.

The current copying collectors for Prolog still have some drawbacks. One drawback is that most simple copying collectors can only do *major* collections, collecting the full heap at once. Since the Prolog program is stopped during the collection cycle, this leads to big pause times for applications with big heaps. Some applications however have timing constraints and require these pause times to be small. Because of this, reducing pause times has always been an important topic in garbage collection. In this work we reduce the pause times by collecting only part of the heap during each collection cycle. This is called incremental collection. An extra benefit of incremental collection in the case of copying collection is that the *to space* is smaller. Most copying collectors are semi-space copying collectors and they need both a *from space* for allocation and a *to space* for collection. The *to space* must always be as big as the *from space*. This means however that only half of the allocated space can be used as useful heap space. In the case of an incremental collector, the *to space* only needs to be as big as the part to be collected in the next collection cycle.

In section 2 we discuss how we modify the standard heap layout of the WAM. We explain the modifications needed for backtracking and trailing in this new heap layout. We also show how instant reclaiming can be done on parts of the heap that have not been collected. Next, in section 3 the implementation of the incremental garbage collector is discussed. We introduce a write barrier and remembered sets, which are needed to guarantee correct incremental collections. We evaluate the performance of the incremental collector in section 4. Some benchmarks are presented to investigate the time performance as well as the memory usage. In section 5 we discuss possible improvements to the current implementation. We intend to further investigate the issues discussed there. Finally, we conclude with section 6.

The experimental evaluation presented in this paper was performed on a Pentium4 1.7Ghz (256Kb L2 cache) with 768Mb RAM. Timings are given in milliseconds, space measurements in heap cells (4 bytes). The incremental collector has been implemented for hProlog 1.7. hProlog is a successor of dProlog [8] and is meant to become a back-end of HAL [7]. hProlog is based on the WAM, but it differs in the following: it has a separate choicepoint and environment stack, it always allocates free variables on the heap and version 1.7 does not tidy the trail on cut. Experimental results are compared to the standard hProlog 1.7 system. hProlog 1.7 uses by default *optimistic copying* [9] for garbage collection. We will refer to the system with the incremental collector as ***inc_gc***, and to the original system with the *optimistic* collector as ***opt_gc***. For the performance evaluation, we disabled *early reset* in ***opt_gc***; we believe this gives a better comparison between the 2 systems, since currently ***inc_gc*** does not implement *early reset*. We discuss how *early reset* can be adapted to fit into ***inc_gc*** in 5.3.

Our benchmark set consists of the following benchmarks: **chess**, **mqueens**, **browsegc**, **boyergc**, **dnamatchgc**, **takgc** and **serialgc**. Benchmarks **chess** and **mqueens** are taken from [9]. The other benchmarks are taken from [12]. They are classical benchmarks, but have extra parameters to increase the size of the benchmark. This makes them more interesting for testing garbage collector performance. We run them with the following input: **browsegc** (5000), **boyergc** (5), **dnamatchgc** (1000), **takgc** (28,16,8), **serialgc** (1000000).

2 Prolog execution with a modified heap layout

2.1 *bb_heap*: a block-based heap layout

The basis for the incremental garbage collector is a modified heap layout, which is better suited for incremental collections. This heap layout consists of several heap blocks with a fixed number of cells. We will refer to this new heap layout as the ***bb_heap***, and to the standard WAM heap layout as the ***wam_heap***.

In ***bb_heap***, the *logical* heap is an *ordered* set of heap blocks. Some extra data structures are used for the management of these blocks. They keep the blocks chronologically ordered, independently of the address order. This block order keeps all data in the ***bb_heap*** ordered by creation time. The most recent block in the heap is used for allocation; we call this the *current block*. Whenever the *current block* overflows, the heap is expanded: an extra heap block is allocated and added to the heap. All heap blocks, that are part of the heap, are active. Heap blocks can become inactive because of instant reclaiming or garbage collection. Inactive blocks are no longer part of the heap, but they are added to a freelist and can be used for future heap expansion.

Note that the idea to divide the heap in separate blocks is not new: incremental copying collectors as in [10, 5] have a very similar heap layout as the one we present here. One important difference though is that ***bb_heap*** keeps a strict order on the heap blocks.

2.2 Heap blocks and backtracking

2.2.1 Instant reclaiming

Because instant reclaiming allows to recover an unbounded amount of memory at a constant cost, it is important to preserve this property during normal program execution. Instant reclaiming relies on the fact that the order of the heap segments is preserved. The ***bb_heap*** keeps the heap data chronologically ordered (as long as no garbage collection has occurred), and as such also keeps the heap segments ordered. This means that instant reclaiming is possible within a block as well as over block boundaries. Upon backtracking we can deallocate all heap cells allocated after the creation of the most recent choicepoint (all heap cells belonging to the topmost heap segment). If the topmost segment consists of several blocks, then all blocks that contain cells belonging to that segment exclusively can be freed. In the block where this segment starts, we can also reclaim that part of the block belonging to that segment.

2.2.2 Trailing

Upon backtracking to a certain choicepoint, all bindings done after the creation of that choicepoint need to be undone. Trailing is the WAM mechanism which remembers these bindings. We only want to record the relevant bindings on the trail: the binding of variables older than the current choicepoint; this is called conditional trailing.

In the ***wam_heap*** this conditional trailing is easily done by comparing the address of the variable which is going to be bound and the heap pointer in the topmost choicepoint, as shown in the following pseudo-code.

```
if (CellPtr < BH) trail(CellPtr);
```

If the variable is older (smaller address) than the pointer in the choicepoint (points to higher address), then the binding should be trailed. This relies on the fact that the continuous ***wam_heap*** grows towards higher memory addresses.

In the ***bb_heap***, things get a little tougher because we cannot rely on the fact that a higher memory address corresponds to a more recent creation time. As long as a block has not been garbage collected, higher memory addresses still correspond to newer cells for cells within that block. For cells in different blocks, we need to know whether one block is older than another one. We do this by giving each block a timestamp when adding it to the heap. Blocks with smaller timestamps are older than blocks with bigger timestamps. Timestamps are preserved during garbage collection. The conditional trailing then looks like the following pseudo-code.

```
if ((same_block(CellPtr,BH) && (CellPtr < BH))
    || (!same_block(CellPtr,BH) && older_block(CellPtr,BH)))
   trail(CellPtr);
```

If the variable and the heap pointer in the choicepoint belong to the same block, then we can use the address order; if not, then we compare the age of the blocks.

A common optimization to reduce trail usage is the following. When a variable is bound to another variable, the youngest cell is always bound to the oldest cell. This reduces the chance that the binding needs to be trailed. This optimization is currently not implemented in the ***bb_heap*** system, but we intend to investigate this in the future.

2.3 Overhead due to heap layout

The more expensive trailing mechanism and the management of the data structures involved in the ***bb_heap*** incur a certain run-time cost, even when garbage collection is not needed or used by the program. To measure this cost, we compare the ***wam_heap*** and the ***bb_heap*** on a number of benchmarks. Both heap layouts are implemented for hProlog 1.7. The ***wam_heap*** system is always started with a heap big enough to run a particular benchmark without needing garbage collection. We configured the ***bb_heap*** system for different heap block sizes (the block size is mentioned in table 1 in bytes). The ***bb_heap*** systems are started with a heap consisting of one block. When the *current block* overflows, an extra block is added to the heap and used for new allocations. No garbage collection is performed. We did not include the benchmarks **mqueens** and **chess** here, since they can not run without garbage collection.

Table 1 shows the time needed to run a particular benchmark with each system. To isolate the performance loss during the execution of a program, we measure the time needed for running the benchmark only. Startup time and allocation of heap space (time spent in malloc and in the UNIX system call mmap) are not considered. Management of our own data structures related to the heap blocks is included in the timings. The results are also shown in a graph in figure 1. The graph shows the performance of the ***bb_heap*** relative to the ***wam_heap***.

We observe the following:

- The benchmarks show that there is some overhead in the ***bb_heap*** systems, but the overhead is generally small with a maximum of 6% for **boyergc** in the ***bb_heap*** 2Mb system.

	wam_heap	***bb_heap*** 2Mb	***bb_heap*** 4Mb	***bb_heap*** 8Mb	***bb_heap*** 16Mb
browsegc	5319	5571	5557	5397	5121
boyergc	9074	9644	9481	9416	9177
dnamatchgc	2414	2488	2508	2499	2446
takgc	1380	1444	1435	1430	1412
serialgc	7725	7635	7622	7600	7448

Table 1: Overhead of the heap layout

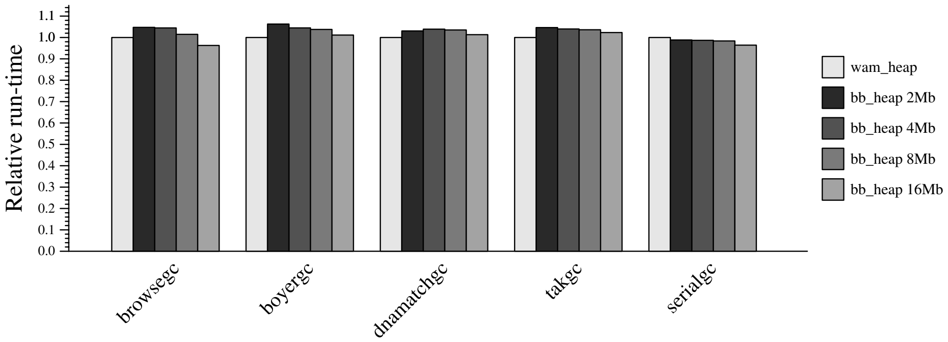


Figure 1: Overhead of the heap layout

- Smaller block sizes in the ***bb_heap*** systems result in more overhead. For a given benchmark, ***bb_heap*** systems with a smaller block size have more blocks; consequently there is more overhead due to the switching of *current block* upon backtracking. Also there are more *current block* overflows, upon which a new block has to be added to the heap.

- For **dnamatchgc**, ***bb_heap*** 2Mb is faster than ***bb_heap*** 4Mb; the difference however, is negligible and is probably due to slight variations in the benchmark conditions or cache effects. A similar reason applies to ***bb_heap*** 16Mb being slightly faster than ***wam_heap*** for **browsegc**.

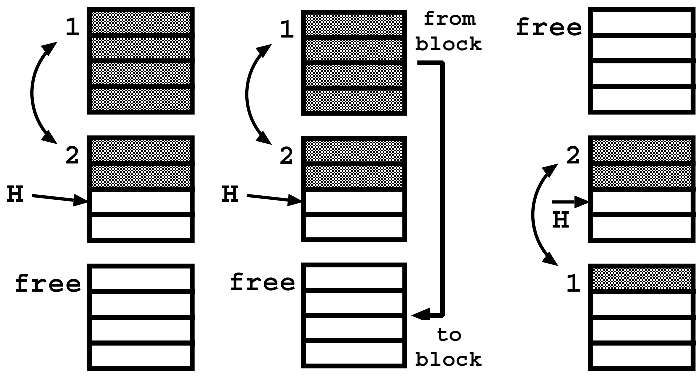
- **serialgc** runs systematically faster in the ***bb_heap*** systems. We suspect this is due to cache effects. With a cache simulator, we found that the ***bb_heap*** systems have more data accesses than the ***wam_heap*** system, but fewer cache misses (up to 10% less L2 cache misses).

3 Incremental collection with heap blocks

3.1 Basic principle

Incremental collection in the ***bb_heap*** is done by collecting one heap block during each collection cycle. Since the garbage collector is based on a copying collector, a *to space* is needed as large as the *from space*. This means that one heap block should always be reserved as *to space*.

Figure 2 shows what happens during garbage collection. Before the collection, in figure 2(a), the ***bb_heap*** consists of 3 blocks: block 1 (the oldest block), block 2 (the *current block*) and one block currently not in use. A garbage collection collects one block (we call this the *from block*) and copies all its live cells to a free block (we call this the *to block*). In this example block 1 is the *from block*. During the collection (figure 2(b)) block 1 is collected and all its live data is copied to the



(a) Before gc (b) During gc (c) After gc

Figure 2: Incremental garbage collection in the ***bb_heap***

free block. Finally in figure 2(c) the *to block* becomes the new block 1, and this block is linked to block 2.

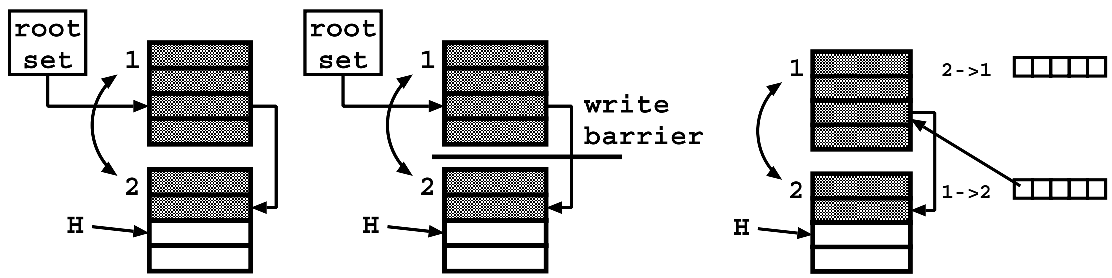
Note that if the *from block* contains garbage (block 1), then the *to block* still contains some free space at the end of the collection. Each heap block has a pointer to indicate this. If the next block (block 2) is collected in a subsequent collection, its live data can be copied to this free space (top of block 1). When all free space is used, the copying is continued with a free block.

The free space left in a *to block* can also be used for subsequent allocation of new data. However, since the *to block* inherits the timestamp from the *from block*, this means that newly allocated data in the free space of the *to block* inherits this timestamp. In case the collected block is different (older) from the *current block*, this can lead to unnecessary trailing of the newly allocated variables, and it makes instant reclaiming more difficult. Therefore our allocation policy never allocates new data in the *to block* if the collected block is different from the *current block*.

3.2 Write barriers and remembered sets

During an incremental collection we want to collect one heap block independently from the other blocks. During the collection cycle, all live data found in that block (the *from block*) is copied to a new block (the *to block*). The *root set* is defined as a set of references, which are known to point to live cells in the heap. In the WAM, the root set consists of references found in the environments, the choicepoints, the trail and the argument registers; all these memory areas contain references to (live) cells on the heap.

However, the root set alone is not sufficient to find all live cells in a heap block without scanning other blocks. In the ***bb_heap***, it is possible that a cell in a block is not referenced by any element of the root set, although it can be referenced by a live cell in another block. An example of this is shown in figure 3(a). The cell is live, because it is indirectly referenced by the root set. At collection time, we want to know which cells in other blocks have references to cells in the *from block*, without scanning all other blocks. We call references from cells in one block (the *source block*) to cells in another block (the *target block*) *inter-block* references; references to cells in the same block are called *intra-block* references. We need a mechanism to remember the *inter-block* references. This can be done with a write barrier and remembered sets.



(a) Indirect reference (b) Write barrier (c) Remembered sets

Figure 3: Write barriers and remembered sets

3.2.1 Write barrier

A write barrier is a mechanism to monitor write operations. Each time a value is about to be written in a memory cell, this action is intercepted by the write barrier. This is a more general mechanism, and is for example also used for trailing. The write barrier for trailing checks whether something is written in a cell older than the current choicepoint, and in that case, it puts the relevant cell on the trail. In this incremental collector, an extra write barrier checks whether an *inter-block* reference is created (figure 3(b)) and in that case, remembers it by recording it in a remembered set (figure 3(c)).

A more elaborate discussion about write barriers and their implementation can be found in [11]. Write barrier implementations can be software-only or use the virtual memory hardware (protecting pages for writing). We implemented a software-only write barrier. The code for the write barrier is only added for assignments where the creation of *inter-block* references is possible, and has been optimized as described in [6].

3.2.2 Remembered sets

Remembered sets are collections of *inter-block* references. Several configurations are possible for the organization of these sets. One commonly used option is to have one remembered set for each block; the write barrier puts all references to cells in a certain block in that block's remembered set. Another option is to have remembered sets for each combination of *source block* and *target block*. Our implementation uses the latter (figure 3(c)). This configuration has 2 important advantages. First, during garbage collection only the remembered sets that have the *from block* as *target block* need to be scanned. Second, after garbage collection, all old entries in the remembered sets should be removed; this corresponds to removing all remembered sets that have the *from block* as either *source block* or *target block*. Note that the write barrier remains active during garbage collection: *inter-block* references newly created during garbage collection are added to new or existing remembered sets.

3.3 Overhead due to write barriers and remembered sets

To investigate the effect of the write barrier on program run-time, we compare the ***wam_heap***, the ***bb_heap*** (without write barrier) and the ***bb_heap***$_{wb}$ (***bb_heap*** with write barrier and remembered sets). Again we present ***bb_heap***$_{wb}$ systems for different sizes of the heap blocks. The same settings apply as in section 2.3. None of the systems do garbage collection.

Table 2 contains the timings for the benchmarks for each ***bb_heap***$_{wb}$ system. The maximal size of the remembered sets (number of entries) is also included. Figure 4 shows the performance of the ***bb_heap***$_{wb}$ systems relative to the performance of the ***wam_heap*** system. The graph also includes the timings for the ***bb_heap*** systems, as presented in 2.3.

	wam_heap	***bb_heap***$_{wb}$ 2Mb		***bb_heap***$_{wb}$ 4Mb		***bb_heap***$_{wb}$ 8Mb		***bb_heap***$_{wb}$ 16Mb	
	t_{tot}	t_{tot}	m_{remset}	t_{tot}	m_{remset}	t_{tot}	m_{remset}	t_{tot}	m_{remset}
browsegc	5319	6404	1042695	6079	782547	5620	0	5668	0
boyergc	9074	9949	1115	9837	606	9769	316	9716	176
dnamatchgc	2414	2598	217	2588	120	2578	50	2571	11
takgc	1380	1465	0	1462	0	1454	0	1434	0
serialgc	7725	9114	22891761	9132	22891060	9031	22890701	9054	22395214

Table 2: Overhead of the write barrier and remembered sets

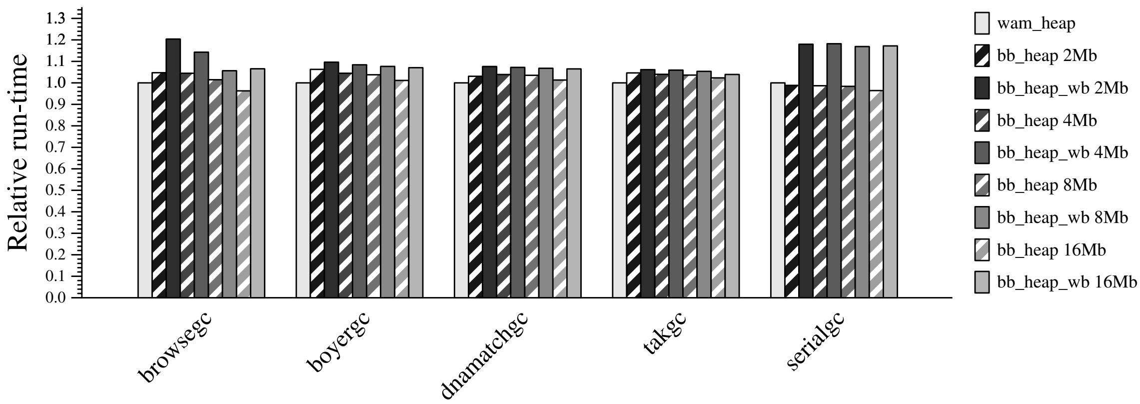


Figure 4: Overhead of the write barrier and remembered sets

From the figures we observe that:

- The write barrier gives an extra overhead; the total overhead of the ***bb_heap***$_{wb}$ systems compared to the ***wam_heap*** system can be as high as 20% (**browsegc** in ***bb_heap***$_{wb}$ 2Mb).
- Benchmarks (**boyergc**, **dnamatchgc**, **takgc**), where most references are very local (the reference and the data it points to are very close to each other in the heap), experience a rather small overhead from the write barrier: in these cases ***bb_heap***$_{wb}$ has an overhead of about 5% compared to ***bb_heap***. The reason is that although the write barrier is triggered, it rarely needs to put references in the remembered sets.
- **serialgc** has a very high overhead in the ***bb_heap***$_{wb}$ systems. The benchmark first initializes some data, and afterwards builds a tree with references to this data. All newer data contains references to the oldest data on the heap. There is a lot of write barrier activity and compared to **boyergc**, **dnamatchgc** and **takgc** the write barrier traps a higher percentage of *inter-block* references, which have to be inserted into the remembered sets.

- **browsegc** is a benchmark where most references are very local, still it experiences very high overhead from the write barrier. We didn't find a good explanation for this behaviour. Cache effects might be the reason.
- The size of the remembered sets becomes smaller as the block size increases. This is what we intuitively expect, because moving from small blocks to bigger blocks, some *inter-block* references become *intra-block* references and as such, there will be less entries in the remembered sets.

4 Experimental results of the incremental garbage collector

The incremental garbage collector for hProlog consists of a ***bb_heap***$_{wb}$ system and the *optimistic copying* algorithm which is used for garbage collection of the heap blocks.

We plan to investigate garbage collection policies in the future (as discussed in section 5.4), but currently we use the following. Garbage collections are always (and only) triggered when the *current block* overflows. The collector then collects the *current block*. If more than 70% of the cells from that block survive the collection, we deem it not worthwhile to collect that block any longer and the next time a garbage collection is triggered, a new block will be added to the heap and used for future allocations. This is a very simple generational policy.

We compare several ***inc_gc*** systems (with different block sizes) with the standard hProlog 1.7 system, ***opt_gc***. We give ***opt_gc*** always an initial heap size of 8Mb. The garbage collection policy in ***opt_gc*** is as follows: a garbage collection is triggered whenever the heap is full, all collections are *major* collections (they collect the whole heap at once) and ***opt_gc*** doubles the heap size whenever more than 70% of the heap survives a collection.

Table 3 contains the results of benchmarks with ***inc_gc***. We do not present all benchmarks here, we left out the ones that are too small to be interesting concerning heap usage. We measured the following items: $\mathbf{t_{gc}}$ (total time spent on garbage collection), $\mathbf{t_{tot}}$ (total run-time, including t_{gc}), **# gcs** (number of garbage collections), $\mathbf{t_{min}/t_{avg}/t_{max}}$ (minimum/average/maximum pause time, time needed for one garbage collection cycle), $\mathbf{mem_{tot}}$ (total memory usage for heap and remembered sets, including *to space*) and $\mathbf{mem_{remset}}$ (space usage of the remembered sets). Time measurements are in milliseconds, space measurements in heap cells.

We observe the following:

- The maximum pause time in the ***inc_gc*** systems is always smaller than in ***opt_gc***; the difference varies from a factor of 1.5 (**chess** and ***inc_gc*** 16Mb) up to a factor of 35 (**serialgc** and ***inc_gc*** 2Mb). Within the ***inc_gc*** systems, smaller block sizes give smaller pause times. The reason for this is that the amount of work for the incremental collector increases with the size of its blocks. The relative difference between pause times across different block sizes depends on the size of the root set; the relative difference will be smaller as the root set becomes bigger. This can be seen for **mqueens**, where the root set is very small (big difference in pause times), and **chess**, where the root set is large.
- The number of garbage collections in ***opt_gc*** is in most cases lower than in ***inc_gc***. This is to be expected since the collections in ***opt_gc*** are *major* and collect the whole heap, while the collections in ***inc_gc*** are *minor* and collect only part of the heap. For the ***inc_gc*** systems, there are also more collections as the block size goes down.
- Total garbage collection time in ***inc_gc*** is sometimes smaller, sometimes bigger than in ***opt_gc***. A priori, we expect garbage collection time in ***inc_gc*** systems to be higher, since

		opt_gc	***inc_gc*** 2Mb	***inc_gc*** 4Mb	***inc_gc*** 8Mb	***inc_gc*** 16Mb
chess	t_{gc}/t_{tot}	1975/10780	5260/14443	3804/12860	1667/10840	1016/9823
	# gcs	6	34	19	7	4
	$t_{min}/t_{avg}/t_{max}$	3/329/656	0/154/290	0/200/320	0/238/353	0/254/426
	mem_{tot}	10702298	8771802	9184855	11195751	12898775
	mem_{remset}		907482	796247	709991	315863
mqueens	t_{gc}/t_{tot}	9663/71743	10463/80213	10555/76943	10302/75910	11379/78433
	# gcs	136	1291	646	321	158
	$t_{min}/t_{avg}/t_{max}$	3/71/220	0/8/24	0/16/47	3/32/87	10/72/146
	mem_{tot}	31996008	15530340	16034702	17062597	22308854
	mem_{remset}		325988	306062	285381	1337334
boyergc	t_{gc}/t_{tot}	5365/14410	6326/16243	3899/13763	2633/12360	1953/11560
	# gcs	14	108	54	25	12
	$t_{min}/t_{avg}/t_{max}$	80/383/1160	10/58/107	27/72/117	50/105/147	87/162/230
	mem_{tot}	31994970	24117913	23068961	23068832	25165889
	mem_{remset}		665	289	160	65
serialgc	t_{gc}/t_{tot}	4730/11930	3620/16400	2807/13993	2488/11703	1614/11000
	# gcs	9	84	42	22	10
	$t_{min}/t_{avg}/t_{max}$	36/525/2544	10/43/74	27/66/97	60/113/166	114/161/200
	mem_{tot}	63996002	49518269	48372865	49321568	42853895
	mem_{remset}		12293821	11672705	11572832	9299463

Table 3: Benchmark results

they have more collections and (during the benchmark) scan the root set more often than ***opt_gc***. However, it is difficult to draw conclusions from these timings. First of all, garbage collection in different systems is triggered at different times. And second, ***inc_gc*** and ***opt_gc*** use a different garbage collection policy; which policy is best, completely depends on the characteristics of the benchmark.

- In most cases, the ***inc_gc*** systems have a lower memory usage than the ***opt_gc*** system. This is what we expected because of the smaller *to space*. In some cases (**chess** in ***inc_gc*** 8Mb and 16Mb) ***inc_gc*** needs more memory. One reason for this is probably that ***inc_gc*** keeps more cells live than ***opt_gc***, because it supposes that all cells in the remembered sets are live, while this is not always true.

- In the ***inc_gc*** systems, the remembered sets become smaller as the block size increases (except for **mqueens** in ***inc_gc*** 16Mb). As already noticed in section 3.3, this is because moving from small blocks to bigger blocks, some *inter-block* references will become *intra-block* references. However this is only valid in systems without garbage collection. In the presence of garbage collection, systems with different block sizes trigger collections at different times. Consequently, data may be placed in different locations on the heap. A data structure like a tree may be placed within one block in one system, while it is spread over 2 heap blocks in another system (and creates a lot of *inter-block* references in that case). Something like this might have happened with **mqueens** in ***inc_gc*** 16Mb. We intend to further investigate the relation between block size and remembered set usage in the future.

- The total run-time in ***inc_gc*** systems with small block sizes is always bigger than in ***opt_gc***. This is partly due to the overhead at run-time, and partly due to the time taken for garbage collection (a lot of collections, that all scan the whole root set).

The benchmarks show that the incremental garbage collector, ***inc_gc***, succeeds to obtain substantially lower pause times, and that it generally has a lower memory usage than ***opt_gc***.

5 Future enhancements

5.1 Tidy trail

Tidy trail is a technique to recover trail space upon cut. When the program executes a cut, it cuts away one or more choicepoints; this can render some entries on the trail useless. *Tidy trail* compacts the trail by removing these entries. Many Prolog systems implement *tidy trail*, but hProlog 1.7 doesn't. For certain programs however, this would drastically reduce trail space usage (for example in **boyergc**). We think this is an important optimization for ***inc_gc***, because the trail is part of the root set, which is scanned at each collection cycle. *Tidy trail* makes the root set smaller, and therefore makes collections cheaper. We plan to implement *tidy trail* for both the ***opt_gc*** and ***inc_gc*** systems and investigate its effect on performance.

5.2 Remembered sets

Currently we use arrays (mainly for simplicity) for the remembered sets and backtracking does not remove entries from the remembered sets. As a result, remembered sets can contain duplicate entries and our collector must take these into account. We plan to investigate the removal of entries from the remembered sets on backtracking further: a better datastructure - like linked lists or hashtables - needs to be introduced.

5.3 Early reset

During garbage collection, it is possible to reset certain bindings recorded on the trail (or for a value-trail to reset the recorded cells to the value they would get upon backtracking); this is called *early reset*. With *early reset*, the garbage collector takes over some of the untrailing work that would be done upon backtracking. This is beneficial because trail entries, subject to *early reset*, can be removed from the trail. Because of this, *early reset* can lead to a smaller root set, just like *tidy trail*.

Early reset is only possible for cells referenced by the trail, which are not used in the forward execution of the program. For each choicepoint, for which we can compute the forward execution and the corresponding set of reachable data, we can apply *early reset* on the accompanying trail segment (that part of the trail that would be untrailed upon backtracking to that specific choicepoint). It is possible to determine the reachable data from a certain choicepoint and apply *early reset* to the appropriate trail entries during garbage collection. A more in-depth discussion about this technique can be found in [2, 3].

Early reset relies on the fact that we can find all live data in the forward execution of any choicepoint. This is done by recursively marking or forwarding all cells from the root set which are reachable through the current environment. Cells, that are not reachable through the current environment are eligible for early reset; they will not be used in the forward execution, so we can safely reset them to the value they will get upon backtracking.

In a ***bb_heap***$_{wb}$ system, in which we do incremental collections, *early reset* is not possible without some modifications. If we want to find all cells reachable in the forward execution, then we should also scan (and mark) cells in other blocks of the heap, because they might have indirect references to cells in the block we are about to collect. An approximation of *early reset* is possible however, we call it ***partial*** *early reset*. As seen in section 3.2, remembered sets contain all references from other blocks of the heap to the *current block*. Before scanning cells in the reachable environments, first we start forwarding from references found in the remembered sets. Since this assumes that all cells referenced from other blocks are live and reachable in the forward execution, this is a conservative approximation of the real set of cells reachable in the forward execution. After forwarding cells referenced through the remembered sets, *early reset* can be done as with a standard heap layout. However, because *partial early reset* is less precise (more conservative), and because we can only do early reset for cells in the *current block*, it will gain less space on the trail as normal *early reset* would do. We intend to implement *partial early reset* and see how it compares to standard *early reset*.

5.4 Garbage collection policy

The garbage collector uses a garbage collection policy to decide about when and how to collect. To initiate a garbage collection, several kinds of triggers can be used. We could use the following criteria as triggers:

- the *current block* is full
- a certain amount of the current heap block is in use
- an amount of allocation has been done since the previous collection
- an amount of time has passed since the previous collection
- the size of certain remembered sets hits a threshold

The current implementation only triggers garbage collection when the *current block* overflows.

The policy also decides which heap block to collect at a certain collection. Several options are available for this. A generational policy would define a number of generations and try to frequently collect blocks in the youngest generations, and only rarely collect blocks in the oldest generations. We could also use a policy to mimic older-first garbage collection. Such a policy would start by collecting the oldest blocks and collect more recent blocks in subsequent collections; in this way it tries to avoid to collect very young data. ***inc_gc*** has a very simple generational policy currently and we plan to investigate more complex policies in the future.

6 Conclusion

We presented an incremental copying collector for WAM-based Prolog systems. The collector relies on the ***bb_heap***, a heap layout different from the standard WAM heap layout. Due to this new heap layout, a number of modifications to the standard WAM were needed. The final system with the incremental collector has decent performance, as shown in a number of benchmarks. In general, the collector substantially lowers pause times and has better memory usage. We also proposed a number of enhancements to the current implementation, which we intend to implement and investigate in the future.

References

[1] H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990 See also: http://www.isg.sfu.ca/~hak/documents/wam.html.

[2] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, June 1988.

[3] Y. Bekkers, O. Ridoux, and L. Ungaro. Dynamic Memory Management for Sequential Logic Programming Languages. In Y. Bekkers and J. Cohen, editors, *Proceedings of IWMM'92: International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 82–102. Springer-Verlag, Sept. 1992.

[4] J. Bevemyr and T. Lindgren. A simple and efficient copying Garbage Collector for Prolog. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 88–101. Springer-Verlag, Sept. 1994.

[5] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, PLDI'02, Berlin, June, 2002*, volume 37(5) of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[6] S. M. Blackburn and K. S. McKinley. In or out?: putting write barriers in their place. In *Proceedings of the third international symposium on Memory management*, pages 175–184. ACM Press, 2002.

[7] B. Demoen, M. García de la Banda, W. Harvey, K. Mariott, and P. Stuckey. An overview of HAL. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 174–188. Springer, 1999.

[8] B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[9] B. Demoen, P.-L. Nguyen, and R. Vandeginste. Copying garbage collection for the WAM: to mark or not to mark ? In P. Stuckey, editor, *Proceedings of ICLP2002 - International Conference on Logic Programming*, number 2401 in Lecture Notes in Computer Science, pages 194–208, Copenhagen, July 2002. ALP, Springer-Verlag.

[10] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *Proc. Int. Workshop on Memory Management*, number 637, pages 388–403, Saint-Malo (France), 1992. Springer-Verlag.

[11] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic memory management.* John Wiley, 1996 See also http://www.cs.ukc.ac.uk/people/staff/rej/gcbook/gcbook.html.

[12] X. Li. Efficient Memory Management in a merged Heap/Stack Prolog Machine. In *Proceedings of the 2nd ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'00)*, pages 245–256. ACM Press, 2000.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

[14] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

Smodels with CLP: a Treatment of Aggregates in ASP

Enrico Pontelli   Tran Cao Son   Islam Elkabani

Department of Computer Science
New Mexico State University
{epontell,tson,ielkaban}@cs.nmsu.edu

Abstract

We present the development of a system that integrates an engine for computation of answer sets with an engine for constraint solving. The integrated system allows us to provide a flexible and effective management of aggregate functions during answer set computation.

1 Introduction

In recent years we have witnessed a renovated interest towards the development of practical, effective, and efficient implementations of different flavors of logic programming. A significant deal of research has been invested in the development of systems that support logic programming under *answer set semantics*, favoring the creation of a novel programming paradigm, commonly referred to as *Answer Set Programming (ASP)*. Many practical systems have been recently proposed to support execution of ASP [14, 7, 1, 11]. The logic-based languages provided by these systems offer a variety of syntactic structures, aimed at supporting the requirements arising from different application domains. Our objective is to introduce different types of *aggregates* in ASP. Database query languages (e.g., SQL) use aggregate functions—such as *sum*, *count*, *max*, and *min*—to obtain summary information from a database. Aggregates have been shown to significantly improve the compactness and clarity of programs in various flavors of logic programming [10, 6]. We expect to gain similar advantages from the introduction of different forms of aggregations in ASP. In the next examples, we demonstrate the use of aggregates in ASP.

Example 1 *Consider a job scheduling problem, where we have a total number of machines (*`resource(Max)`*) and a number of jobs. Each job uses one machine and lasts for a certain duration (*`duration(Job,Time)`*). We can define a predicate* $active(Job, Time)$ *and use the aggregate count to calculate the number of active jobs at the time instance* $Time$*. The number of active jobs cannot be greater than the number of machines:*

active(Job,Time)	:-	*time(Time), duration(Job, Len), start(Job,Init),*
		Time >= Init, Time < Init+Len.
	:-	*time(T), resource(Max), count(T, active(Job,T)) > Max.*

Example 2 ([12]) *Let* $owns(X, Y, N)$ *denote the fact that company* X *owns a fraction* N *of the shares of the company* Y*. We say that a company* X controls *a company* Y *if the sum of the shares it owns in* Y *together with the sum of the shares owned in* Y *by companies controlled by* X *is greater than half of the total shares of* Y*.*[1]

[1]For the sake of simplicity we omitted the domain predicates required by *smodels*.

control(X, X, Y, N)	:-	*owns(X,Y,N).*
control(X, Z, Y, N)	:-	*control(X,Z), owns(Z,Y,N).*
fraction(X,Y,N)	:-	*sum(M,control(X,Z,Y,M)) = N.*
control(X,Y)	:-	*fraction(X,Y,N), N >0.5.*

A significant body of research has been developed in the database and in the constraint programming communities exploring the theoretical foundations and, in a more limited fashion, the algorithmic properties of aggregation constructs in logic programming (e.g. [10, 16, 12, 5]). More limited attention has been devoted to the more practical aspects related to computing in logic programming in presence of aggregates. In [2], it has been shown that aggregate functions can be encoded in ASP (e.g., example 1 above). The main disadvantage of this proposal is that the obtained encoding contains several intermediate variables, thus making the grounding phase quite expensive in term of space and time. Recently, a number of proposals to extend logic programming with aggregate functions have been developed, including work on the use of aggregates in the ASET system [8], extensive work on sets and grouping in logic programming [15, 6], and the excellent implementation of aggregates in the *dlv* system [4]. of A-Prolog enriched with aggregation on sets. The specific approach proposed in this work accomplishes the same objectives as [4, 8]. The novelty of our approach lies in the technique adopted to support aggregates. Following the spirit of our previous efforts [6, 3], we rely on the integration of different *constraint solving* technologies to support the management of different flavors of sets and aggregates. In this paper, we describe a back-end inference engine, obtained by the integration of *smodels* with a finite-domain constraint solver, capable of executing *smodels* program with aggregates. The back-end is meant to be used in conjunction with front-ends capable of performing high-level constraint handling of sets and aggregates (as in [6]). We will refer to that system as *smodels–ag* hereafter.

2 The Language

Now we will give a formal definition for the syntax and semantics of the language accepted by the *smodels-ag* system. This language is an extension to the language accepted by the *smodels* system by aggregate functions.

2.1 Syntax

The input language accepted by our system is analogous to the language of smodels with the exception of a new class of literals - *the aggregate literals.*

Definition 1 *An* smodels-ag *system* intentional set *is a set of the form* $\{X, Goal[X, \bar{Y}]\}$.
An smodels-ag *system* intentional multiset *is a multiset of the form* $\{\{X, Goal[X, \bar{Y}]\}\}$.
where X *is the grouped variable while* $\bar{Y}$ *are the existentially quantified variables. i.e.* $\{X : \exists\ \bar{Y}$ *s.t* $Goal[X, \bar{Y}]$ *is true*$\}$.
and $Goal[X, \bar{Y}]$ *is an atom or an expression of the form:*

- $atom(X)$ *(where* $\bar{Y}$ *is empty).*
- $atom1(X, \bar{Y})$: $atom2(\bar{Y})$ *(where* $atom2(\bar{Y})$ *is the domain predicate of* $\bar{Y}$*).*

Definition 2 *An* aggregate function *is of the form* $aggr(S)$*, where* S *is either an intensional set or an intensional multiset, and* $aggr$ *is one of the following functions: count, sum, min, max. An* aggregate atom *is of the form* $aggr(S)$ **op** *Result, where* **op** *is one of the relational operators drawn from the set* $\{=, != , <, >, <=, >=\}$ *and Result is either a variable or a numeric constant.*

The variables $X, \bar{Y}$ are locally quantified within the aggregate. At this time, the aggregate literal cannot play the role of a domain predicate—thus other variables appearing in an aggregate literal (e.g., Result) are treated in the same way as variables appearing in a negative literal.

Definition 3 *An* smodel-ag *rule is in the form:*

$$B \leftarrow L_1, \ldots, L_n$$

where B is a positive literal, $L_1, \ldots, L_n$ are either positive or negative aggregate or non-aggregate literals.

For simplicity, we require the body of each *smodel-ag* rule to contain at most one aggregate.

Definition 4 *An* smodel-ag *program is a set of* smodel-ag *rules.*

In *smodels-ag*, we have opted for relaxing the stratification requirement present in [4, 8], which avoids the presence of recursion through aggregates. The price to pay is the possibility of generating non-minimal models [6, 10]; on the other hand, the literature has highlighted situations where stratification of aggregates prevents natural solutions to problems [12, 5].

Example 3 *Consider the following program:*

$$\begin{array}{lll} p(1). & p(2). & p(3). \\ q & :- & sum\{X, q(X)\} > 10. \\ p(5) & :- & q. \end{array}$$

This program contains recursion through aggregates. It has the two answer sets $A_1 = \{p(1), p(2), p(3)\}$ and $A_2 = \{p(1), p(2), p(3), p(5), q\}$. It is obvious that the model A_2 is not a minimal model, which is a natural result due to the recursion through aggregate atoms.

2.2 Semantics

Now we will provide the semantics for the language. In particular, we are interested in the stable model semantics [9] of the language based on the interpretation of the aggregate atoms. First, we will start by defining the domains over which aggregate functions are defined and *smodels-ag* program variables are assigned.

Definition 5 *Let P be an* smodels-ag *program, then U_P is the set of all constants appearing in P. Let P be an* smodels-ag *program, then $\mathcal{U}_P$ is the set of all possible multisets over constants appearing in U_P.*

Let P be an smodels-ag *program, then $\mathcal{U}_P^{\mathcal{N}}$ is the set of all possible multisets over natural numbers appearing in U_P. Now we can give the meanings of the aggregate functions as follows:*

- *sum: It is a mapping from $\mathcal{U}_P^{\mathcal{N}}$ to $\mathcal{N}$. It computes the sum of the elements in a multiset.*
- *count: It is a mapping from $\mathcal{U}_P$ to $\mathcal{N}$. It computes the number of the elements in a multiset.*
- *min: It is a mapping from $\mathcal{U}_P^{\mathcal{N}}$ to $\mathcal{N}$. It computes the minimum element in a multiset.*
- *max: It is a mapping from $\mathcal{U}_P^{\mathcal{N}}$ to $\mathcal{N}$. It computes the maximum element in a multiset.*

Definition 6 *(Valuation Function) Let $\mathcal{V}$ denotes the set of variables in program P then $\sigma : \mathcal{V} \mapsto U_P$ is the valuation$function$ that maps variables in P to constants in U_P.*

Definition 7 *(Grounding) Given an atom A and a valuation function σ for variables in A, we define the notion of grounding of A w.r.t σ as follows:*

- *If $A = p(v_1, \ldots, v_n)$ is a non-aggregate atom, then the grounding of $p(v_1, \ldots, v_n)$ w.r.t σ is given by $p(\sigma(v_1), \ldots, \sigma(v_n))$.*
- *If $A = aggr\{X, Goal(X, \bar{Y})\}$ op Result is an aggregate atom, then its grounding w.r.t σ is given by $aggr\{\langle\sigma(X), Goal(\sigma(X), \sigma(\bar{Y}))\rangle\}$ op Result, where $\{\langle\sigma(X), Goal(\sigma(X), \sigma(\bar{Y}))\rangle\}$ is a ground set of pairs.*

The notion of grounding can be extended to rules and to programs. We denote a grounded program P by $Ground(P)$.

Definition 8 *Let P be an* smodels-ag *program, then B_P is the set of all ground non-aggregate atoms which can be formed by using predicate symbols in P with constants from U_P.*

Now we are ready to assign a precise meaning to the syntactic entities of our language (aggregate and non-aggregate atoms) w.r.t an interpretation I. An interpretation I of an *smodels-ag* program P is defined, informally, as a subset of non-aggregate atoms in B_P (i.e, $I \subseteq B_P$).

Definition 9 *An* interpretation *I for an* smodels-ag *program P is a function that maps a grounded atom A in P into its truth value as follows:*

- *If A is a grounded non-aggregate atom then we say that the interpretation of A is true w.r.t I if $A \in I$, otherwise A is $false$ w.r.t I.*
- *If A is a grounded aggregate atom in the form*

$$aggr\{\langle\sigma(X), Goal(\sigma(X), \sigma(\bar{Y}))\rangle\} \text{ op Result}$$

then we say that the interpretation of A is true w.r.t I if $aggr\{\sigma(X) \mid where\ Goal(\sigma(X), \sigma(\bar{Y})) \in I\}$ op Result is true, otherwise A is $false$ w.r.t I, where the interpretation of aggr is a value computed according to the meaning of aggr that we have mentioned above, and the interpretation of op is given by the usual meaning of the relational operators $\{=, !=, <, >, <=, >=\}$.

We define an interpretation I to be a model of a grounded atom A denoted by $I \models A$, if A is true w.r.t I. Similarly, we can extend the definition of $\models$ to conjunctions of atoms. We define an interpretation I to be a model of a grounded rule: $B \leftarrow L_1, \ldots, L_n$, where $L_1, \ldots, L_n$ are either aggregate or non-aggregate literals, if I *models* $L_1, \ldots, L_n$ implies I *models* B. I is a model of a rule if it is a model of each grounding of the rule. We can notice now that the stable models defintion of an *smodel-ag* program might follow directly from [9]. Also, we shall remember that stable models minimality can not be guaranteed in this case, as mentioned earlier.

3 Integrating a Constraint Solver in an Answer Set Solver

We now describe the most relevant aspects of our system. The general idea of our solution is to employ *finite domain constraints* to encode the aggregates in a program.

3.1 Representing Aggregates as Constraints

Now we are going to show how to represent all different kinds of aggregates as finite domain constraints. First, each atom appearing in an aggregate is represented as a variable with domain 0..1; then the whole aggregate is expressed as a constraint involving such variables. This can be shown as in the following subsections.

3.1.1 Count Aggregate

An aggregate atom in the form $count\{X, Goal(X)\}$ *op Result* is represented as a finite domain constraint in the form:

$$X[i_1] \;+\; X[i_2] \;+\; \ldots \;+\; X[i_n] \; con\_op \; Result$$

where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal(X)$, the i's are the indices of the ground atoms in the atom table and *con_op* is the ECLiPSe operator corresponding to the relational operator *op*. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $count\{A, p(A)\}{<}3$ will lead to the constraint

$$X[1]{::}0..1,\; X[2]{::}0..1,\; X[3]{::}0..1,\; X[1]{+}X[2]{+}X[3] \;\#{<}\; 3$$

where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively.

3.1.2 Sum Aggregate

An aggregate atom in the form $sum\{X, Goal(X)\}$ *op Result* is represented as a finite domain constraint in the form:

$$X[i_1] \;*\; v_{i_1} \;+\; X[i_2] \;*\; v_{i_2} \;+\; \ldots \;+\; X[i_n] \;*\; v_{i_1} \; con\_op \; Result$$

where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal(X)$, the i's are the indices of the ground atoms in the atom table, v_i's are the constants instantiating the atom $Goal(X)$ and *con_op* is the ECLiPSe operator corresponding to the relational operator *op*. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $sum\{A, p(A)\}{<}3$ will lead to the constraint

$$X[1]{::}0..1,\; X[2]{::}0..1,\; X[3]{::}0..1,\; X[1]{*}1{+}X[2]{*}2{+}X[3]{*}3 \;\#{<}\; 3$$

where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively.

3.1.3 Max Aggregate

An aggregate atom in the form $max\{X, Goal(X)\}$ *op Result* is represented as a finite domain constraint in the form:

$$maxlist([\; X[i_1] * v_{i_1},\; X[i_2] * v_{i_2}, \quad \ldots,\; X[i_n] * v_{i_n} \;]) \; con\_op \;\; Result$$

where the $X[i]$'s are finite domain constraint variables representing all the ground atoms of $Goal(X)$, the i's are the indices of the ground atoms in the atom table, v_i's are the constants instantiating the atom $Goal(X)$ and *con_op* is the ECLiPSe operator corresponding to the relational operator *op*. E.g., given the atoms $p(1), p(2), p(3)$, the aggregate $max\{A, p(A)\}{<}5$ will lead to the constraint

$$X[1]{::}0..1,\; X[2]{::}0..1,\; X[3]{::}0..1,\; maxlist([X[1] * 1, X[2] * 2, X[3] * 3]) \;\#{<}\; 5$$

where $X[1], X[2], X[3]$ are constraint variables corresponding to $p(1), p(2), p(3)$ respectively.

3.1.4 Min Aggregate

It might seem that the representation of the $min\{X, Goal(X)\}$ *op Result* aggregate atom as a finite domain constraint is analogous to that of the max aggregate with the only difference of using $minlist/1$ instead of $maxlist/1$. This is not absolutely true. We have noticed a problem that might evolve when we represent the min aggregate in the same way as we did with the max aggregate. The problem is that we might have one or more values of the X_i's are set to 0, which are the X_i's that represent ground atoms having false truth values, this might lead to a wrong answer when we compute the minimum value in a list, since the result will be 0 all the time, although the real minimum value could be another value rather than 0 (the minimum value of the v_i's that correspond to the $X[i]$'s representing ground atoms having true truth values). E.g. , given the atoms $p(3), p(4), p(5)$, if we already knew that $p(3)$ and $p(4)$ are true, while $p(5)$ is false, in this case if we use the same representation as the max aggregate in representing the aggregate $min\{A, p(A)\}{<}2$ that will lead to the constraint

$$X[1]{::}0..1,\ X[2]{::}0..1,\ X[3]{::}0..1,\ minlist([X[1]*3, X[2]*4, X[3]*5])\ \#{<}\ 2.$$

This representation is wrong, since in this case the result for this constraint will be true, since the result from applying minlist will be 0 and 0 is less than two, but the correct answer should be false, since the minimum of the values that correspond to ground atoms having true truth value is 3 which is not less than 2. In order to overcome this problem, we have suggested the following representation of the aggregate atom $min\{X, Goal(X)\}$ *op Result* as a finite domain constraint:

$$
\begin{array}{l}
Y[i_1]\ \#=\ (X[i_1]{*}1){+}1,\\
Y[i_2]\ \#=\ (X[i_2]{*}2){+}1,\\
\qquad\quad \vdots\\
Y[i_n]\ \#=\ (X[i_n]{*}n){+}1,\\
\quad element(Y[i_1], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_1]),\\
\quad element(Y[i_2], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_2]),\\
\qquad\quad \vdots\\
\quad element(Y[i_n], [\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ],\ Z[i_n]),\\
\quad minlist([\ Z[i_1],\ Z[i_2],\ \ldots,\ Z[i_n]\ ])\ \ con\_op \qquad Result
\end{array}
$$

where M is a very large constant such that $M > v_i$, for all possible values of i, the Y_i's are selector indices that are used to select a value from the list $[\ M,\ v_{i_1}, \ldots,\ v_{i_n}\ ]$ to be assigned to the $Z[i]$'s by using the fd-library constraint $element/3$ and the $Z[i]$'s are the new list of $X[i] * v_i$ with the exception that each $X[i] * v_i$ that corresponds to an atom with a false truth value is changed to M. E.g. , by applying this treatment for the previous example, we will find that $Z[1]$ is assigned 3, $Z[2]$ is assigned 4 and $Z[3]$ is assigned a large number, say 100000. In this case the result of the constraint $minlist([Z[1], Z[2], Z[3]])\ \#{<}\ 2$ is false, which is a correct answer (since $3 < 2$ is false).

3.2 System Architecture

The overall structure of *smodels-ag* is shown in Fig. 1. The current implementation is built using *smodels (2.27)* and the ECLiPSe (5.4) constraint solver. At this stage it is a prototype aimed at investigating the feasibility of the proposed ideas.

3.2.1 Preprocessing

The Preprocessing module is composed of three sequential steps. In the *first* step, a program – called *Pre-Analyzer* – is used to perform a number of simple syntactic transformations of the input program. The transformations are mostly aimed at rewriting the aggregate literals in a format acceptable by *lparse*. The *second* step executes the *lparse* program on the output of the pre-analyzer, producing a grounded version of the program encoded in the format required by *smodels* (i.e., with a separate representation of rules and atoms). The *third* step is performed by the *Post-Analyzer* program whose major activities are:

- Identification of the dependencies between aggregate literals and atoms contributing to such aggregates; these dependencies are explicitly included in the output file. (The *lparse* output format is extended with a fourth section, describing these dependencies.)
- Generation of the constraint formulae encoding the aggregate; e.g., an entry like *"57 sum(x,use(8,x),3,greater)"* in the atom table (describing the aggregate *sum(X,use(8,X))* $>$ *3*) is converted to *"57 sum(3,[16,32,48],"X16 * 2 + X32 * 1 + X48 * 4 + 0 #> 3")"* ($16, 32, 48$ are indices of the atoms *use(8,_)*).
- Simplification of the constraints making use of the truth values discovered by *lparse*.

3.2.2 Models Computation

The Model Computation module (Fig. 1) is in charge of generating the models from the input program. The module consists of a modified version of *smodels* interacting with an external finite domain constraint solver (in this case, the ECLiPSe solver).

As in *smodels*, each atom in the program has a separate internal representation—including aggregate literals. In particular, each aggregate literal representation maintains information regarding what program rules it appears in. The representation of each aggregate literal is similar to that of a standard atom, with the exception of some additional fields; these are used to store an ECLiPSe structure representing the constraint associated to the aggregate. Each standard atom includes a list of pointers to all the aggregate literals depending on such atom.

4 Data Structures

Now we will describe in more details the modifications done to the *smodels* system data structures, in order to extend the *smodels* system with aggregate functions and make it capable of communicating the ECLiPSe constraint solver.

4.1 Atom

Most of the new data structures that have been added in the new *smodels-ag* system are extension to the class Atom. This is because we are introducing a new type of atoms (aggregate constraint atom) which has its own properties. In order to represent these properties we have augmented the class Atom with the following data structures:

- *bool* **aggregate** : It specifies whether this atom is an aggregate constraint atom or not.
- *Atom *** **dependents**: If this atom is an aggregate constraint, *dependents* is the list of atoms on which this aggregate depends on.
- *int* **numberofdependents**: If this atom is an aggregate constraint, *numberofdependents* is the number of atoms on which this aggregate depends on

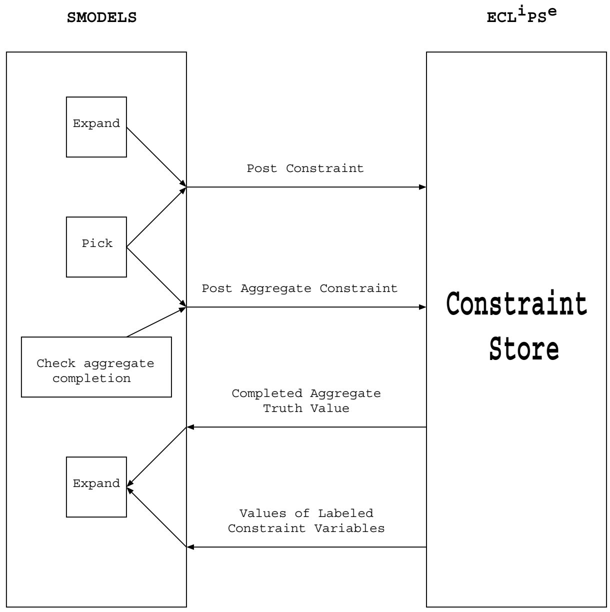


Figure 1: Overall System Structure

- *Atom* ** **constraints**: It is the list of aggregate constraint atoms that depends on this atom.
- *int* **numberofconstraints**: It is the number of aggregate constraint atoms that depends on this atom.
- *int* **met_dependents**: If this atom is an aggregate constraint, *met_dependents* is the number of its dependent atoms which still have unknown truth values.
- *long* **my_index**: It is the index of this atom in the atom table.
- *char* * **constraint_formula**: If this atom is an aggregate constraint atom, *constraint_formula* is the formula of this aggregate constraint atom that will be posted on the ECLiPSe constraint store. (e.g., $X[12]*1 + X[13]*2 + X[14]*3 \;\#< 4$).
- *char* * **dependent_formula**: If this atom is an aggregate constraint atom, *dependent_formula* is the conjunction of the primitive constraint dependents of this aggregate constraint that are already posted on the ECLiPSe constraint store. (e.g., true, $X[12] = 1$, $X[13] = 0$).
- *EC_word* **PosCon**: It is an ECLiPSe data structure that holds the positive constraint that will be posted into the ECLiPSe constraint store. (e.g., X[12] #= 1).
- *EC_word* **NegCon**: It is an ECLiPSe data structure that holds the negative constraint that will be posted into the ECLiPSe constraint store. (e.g., X[13] #= 0).
- *EC_word* **conterm**: It is an ECLiPSe data structure that holds the aggregate constraint that will be posted into the ECLiPSe constraint store.

4.2 Finite Domain Variables

The communication between the *smodels* system and the ECLiPSe is a two-way communication. *smodels* system is posting constraints into the ECLiPSe constraint solver. On the other hand, ECLiPSe is communicating the *smodels* by either sending the truth value of a posted completed aggregate constraint or by sending back values of labeled variables to the *smodels* system as an answer for a posted non-completed aggregate constraint. In order to return an answer of a posted non-completed aggregate constraint to the *smodels*, we need to refer to these ECLiPSe terms (finite

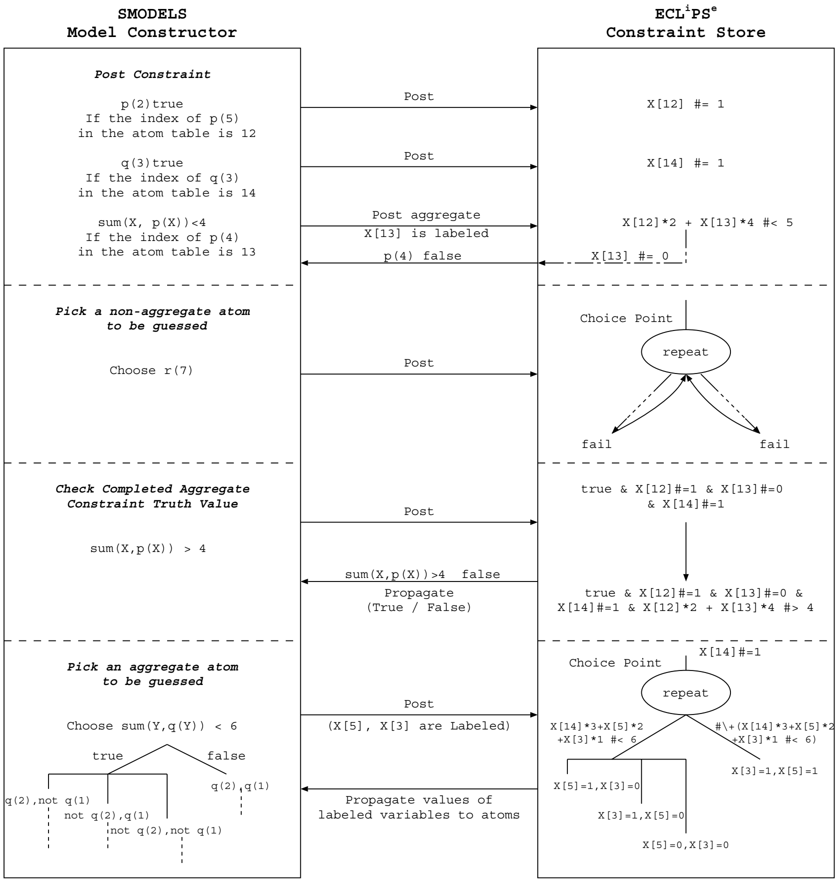


Figure 2: Communication *smodels* to *ECLiPSe*

domain variables) from the *smodels* (C++). This can be done by using the ECLiPSe data types *EC_refs* and *EC_ref*. In our case, we have added the following data structures as global variables in order to handle this situation:

EC_refs **X(n)**: It is an ECLiPSe data structure that holds n references for n ECLiPSe variables. These variables hold the recent values of the ECLiPSe finite domain constraints variables. By this way we can have an easy access from the *smodels* to the values of the labeled variables assigned in the ECLiPSe.

EC_ref ***DigitList**: It is an ECLiPSe data structure that holds a reference to a list of ECLiPSe finite domain variables.

5 Execution Control

In this section we will describe the execution of the system in details. The main flow of execution is directed by *smodels*. In parallel with the construction of the model, our system builds a *constraint store* within ECLiPSe. The constraint store maintains *one conjunction* of constraints, representing the level of aggregate instantiation achieved so far. The implementation of the *smodels-ag* system required changes to almost all the modules of the *smodels* system. During our description for the control of execution, we are going to highlight some of the main changes that have been applied to the *smodels* system modules.

Expand: The goal of the *smodels* expand module is to extend the set of atoms whose truth values are known (true/false) as much as possible. In our *smodels-ag* system we extend the expand module such that each time an aggregate dependent atom is made true or false, a new constraint is posted in the constraint store. If i is the index of such atom within *smodels*, and the atom is made true (false), then the constraint $X[i]\#{=}1$ ($X[i]\#{=}0$) is posted in the ECLiPSe constraint store. (Fig. 2, first two `post` operations). If the ECLiPSe returns EC_fail this means that a conflict is detected (inconsistency), so the control returns to the *smodels* where the conflict is handled. Otherwise, ECLiPSe returns EC_succeed and the control returns to the *smodels* expand module again.

Also since aggregate literals are treated by *smodels* as standard program atoms, they can be made true, false, or guessed. The only difference is that, whenever their truth value is decided, a different type of constraint will be posted to the store—i.e., the constraint representation of the aggregate (Fig. 2, third `post` operation). If the aggregate literal is made false, then a negated constraint will be posted (negated constraints are obtained by applying the #\+ ECLiPSe operator). In this case, ECLiPSe returns an answer for the posted aggregate constraint to the *smodels*, in other words, it returns an instantiation for labeled variables that satisfies the aggregate constraint. This instantiation is interpreted into a truth value for atoms in the *smodels* and then the control returns to the expand module again. If there are more than one answer for the aggregate constraint, the control must return back to the ECLiPSe for backtracking and generating another answer, this happened after the *smodels* computes the stable model containing the previous answer and backtracks for computing another model.

Observe that the constraints posted to the store have an active role during the execution:

- constraints can provide feedback to *smodels* by forcing truth or falsity of previously uncovered atoms (truth value is unknown at that time). E.g., if the constraint $X[12]{*}2 + X[13]{*}4\#{<}5$ is posted to the store (corresponding to the aggregate $sum(X, p(X)){<}4$) and $X[12]\#{=}1$ has been previously posted (e.g., $p(2)$ is true), then it will force $X[3]\#{=}0$, i.e., $p(3)$ to be false (Fig. 2, third `post` operation).
- inconsistencies in the constraint store have to be propagated to the *smodels* computation.

Check aggregate completion: An aggregate literal may become true/false not only as the result of the deductive closure computation of the *smodels expand procedure*, but also because enough evidence has been accumulated to prove its status. E.g., if the truth value of all atoms involved in the aggregate has been established, then the aggregate can be immediately evaluated. In this case, we call the aggregate literal the *completed aggregate*. Every time an aggregate dependent atom is made true or false its aggregate literal is checked for its completion. If the aggregate is completed, then it's constraint representation is posted in the constraint store (Fig. 2, fifth `post` operation). In this case, ECLiPSe returns the truth value of the aggregate constraint to the *smodels*, then the control returns back to the expand module in order to use this knowledge (aggregate literal truth value) in its deductive closure computation. E.g., if the constraint $X[12]{*}2{+}X[13]{*}4\#{>}4$ is posted to the store (corresponding to the aggregate $sum(X, p(X)){>}4$) and $X[12]\#{=}1$ and $X[13]\#{=}0$ have been previously posted (e.g., $p(2)$ is true and $p(4)$ is false), then in this case ECLiPSe returns false as a truth value for the aggregate literal $sum(X, p(X)){>}4$ to the *smodels* (Fig. 2, fifth `post` operation). Similarly, as in the previous cases, in this case also inconsistencies in the constraint store are propagated to the *smodels* for handling conflicts.

Pick: The structure of the computation developed by *smodels* is reflected in the structure of the constraints store (see Fig. 2). In particular, each time *smodels* generates a choice point (e.g., as effect of guessing the truth value of an atom), a corresponding choice point has to be generated in the store. Similarly, whenever *smodels* detects a conflict and initiates backtracking, a failure has to be triggered in the store as well (see Fig. 2, fourth `post` operation). Observe that choice points and failures can be easily generated in the store using the *repeat* and *fail* predicates of ECLiPSe. In our *smodels-ag* system, we have extended the *smodels* pick module to allow aggregate atoms to be picked and its truth value is guessed in the same manner as in the case of non-aggregate atoms. Obviously, aggregate atoms that are picked are non-completed aggregate atoms since, as we mentioned previously, aggregate atoms are checked for their completion every time a dependent atom is made true or false. In this case, the picked aggregate atom is set to true and posted into the constraint store with labeling the unbounded finite domain variables of the aggregate constraint formula. A choice point is generated into the ECLiPSe constraint store and an answer is generated for the labeled variables, then the control returns back to the *smodels* for continuing the current model computation. If a conflict is detected, it is propagated to the ECLiPSe constraint store where a failure is generated to force backtracking to the choice point, then the control returns back to the *smodels* where backtracking takes place and then the aggregate atom is set to false and a negated constraint representation of the aggregate atom is posted. If no conflicts were detected, then the *smodels* will continue the computation of the model and a backtracking will take place for constructing a new model. At this point the control will return back to the ECLiPSe where a new answer is generated (Fig. 2, last `post` operation).

6 Discussion and Conclusions

The prototype implementing these ideas has been completed and used on a pool of benchmarks. Performance is acceptable, but we expect to obtain significant improvements by refining the interface with ECLiPSe. Combining a constraint solver with *smodels* brings many advantages:

- since we are relying on an external constraints solver to effectively handle the aggregates, the only step required to add new aggregates (e.g., *times*, *avg*) is the generation of the appropriate constraint formula during preprocessing;
- the constraint solvers are very flexible; e.g., by making use of Constraint Handling Rules we can implement different strategies to handle constraints and new constraint operators;
- the constraint solvers automatically perform some of the optimizations described in [4];
- it is a straightforward extension to allow the user to declare aggregate instances as *eager*; in this case, instead of posting only the corresponding constraint to the store, we will also post a *labeling*, forcing the immediate resolution of the constraint store (i.e., guess the possible combinations of truth values of selected atoms involved in the aggregate). In this way, the aggregate will act as a generator of solutions instead of just a pruning mechanism.

We believe this approach has advantages over previous proposals. The use of a general constraint solver allows us to easily understand and customize the way aggregates are handled (e.g., allow the user to select eager vs. non-eager treatment); it also allows us to easily extend the system to include new form of aggregates, by simply adding new type of constraints. Furthermore, the current approach relaxes some of the syntactic restriction imposed in other proposals (e.g., stratification of aggregations). The implementation requires minimal modification to the *smodels* system and introduces insignificant overheads for regular programs. The prototype confirmed the feasibility of this approach. Future work includes:

- further relaxation of some of the syntactic restrictions. E.g., the use of labeling allows the aggregates to “force” solutions, so that the aggregate can act as a generator of values; this may remove the need to include domain predicates to cover the result of the aggregate (e.g., the *safety* condition used in *dlv*).

- development of an independent grounding front-end; the current use of a pre-analyzer is dictated by the limitations of *lparse* in dealing with syntactic extensions.

Acknowledgments

The research has been partially supported by NSF grants CCR-9900320, EIA-00220590, CCR-0130887, and CCR-0975279.

References

[1] Y. Babovich and V. Lifschitz. Computing Answer Sets Using Program Completion.

[2] C. Baral. *Knowledge Representation, reasoning, and declarative problem solving*, Cambridge, 2003.

[3] A. Dal Palu' et al. Integrating Finite Domain Constraints and CLP with Sets. *PPDP*, ACM, 2003.

[4] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer. . Aggregate Functions in Disjunctive Logic Programming. IJCAI, 2003.

[5] M. Denecker et al. Ultimate well-founded and stable semantics for logic programs with aggregates. In *ICLP*, Springer. 2001.

[6] A. Dovier, E. Pontelli, and G. Rossi. Intensional Sets in CLP. *ICLP*, Springer Verlag, 2003.

[7] T. Eiter et al. The KR System `dlv`: Progress Report, Comparisons, and Benchmarks. In *KRR*, 1998.

[8] M. Gelfond. Representing Knowledge in A-Prolog. *Logic Programming & Beyond*, Springer, 2002.

[9] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pp. 1070–1080, MIT Press, 1988.

[10] D. Kemp and P. Stuckey. Semantics of Logic Programs with Aggregates. In *ILPS*, 1991.

[11] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of Logic Programs By SAT Solvers. *AAAI'02*.

[12] K. Ross and Y. Sagiv. Monotonic Aggregation in Deductive Databases. *JCSS*, 54, 1997.

[13] K. A. Ross et al. Foundations of aggregation constraints. *TCS*, 193(1–2), 1998.

[14] P. Simons et al. Extending and Implementing the Stable Model Semantics. *AIJ*, 138, 2002.

[15] O. Shmueli et al. Compilation of Set Terms in the Logic Data Language. *JLP*, 12(1/2), 1992.

[16] A. Van Gelder. The Well-Founded Semantics of Aggregation. In *PODS*, ACM Press, 1992.

Implementing Constructive Negation

Susana Muñoz Juan José Moreno-Navarro
susana@fi.upm.es jjmoreno@fi.upm.es

LSIIS, Facultad de Informática
Universidad Politécnica de Madrid
Campus de Montegancedo s/n Boadilla del Monte
28660 Madrid, Spain **

Abstract. Logic Programming has been advocated as a language for system specification, especially for those involving logical behaviours, rules and knowledge. However, modeling problems involving negation, which is quite natural in many cases, is somewhat limited if Prolog is used as the specification / implementation language. These restrictions are not related to theory viewpoint, where users can find many different models with their respective semantics; they concern practical implementation issues. The negation capabilities supported by current Prolog systems are rather constrained, and there is no a correct and complete implementation. In this paper, we refine and propose some extensions to the method of constructive negation, providing the complete theoretical algorithm. Furthermore, we also discuss implementation issues providing a preliminary implementation.

Keywords Constructive Negation, Negation in Logic Programming, Constraint Logic Programming, Implementations of Logic Programming.

1 Introduction

From its very beginning Logic Programming has been advocated to be both a programming language and a specification language. It is natural to use Logic Programming for specifying/programming systems involving logical behaviours, rules and knowledge. However, this idea has a severe limitation: the use of negation. Negation is probably the most significant aspect of logic that was not included from the outset. This is due to the fact that dealing with negation involves significant additional complexity. Nevertheless, the use of negation is very natural and plays an important role in many cases, for instance, constraints management in databases, program composition, manipulation and transformation, default reasoning, natural language processing, etc.

Although this restriction cannot be perceived from the theoretical point of view (because there are many alternative ways to understand and incorporate negation into Logic Programming), the problems really start at the semantic level, where the different proposals (negation as failure (*naf*), stable models, well-founded semantics, explicit negation, etc.) differ not only as to expressiveness but also as to semantics. However,

** This work was partly supported by the Spanish MCYT project TIC2000-1632.

the negation techniques supported by current Prolog compilers are rather limited, restricted to negation as failure under Fitting/Kunen semantics [8] (sound only under some circumstances usually not checked by compilers) which is a built-in or library in most Prolog compilers (Quintus, SICStus, Ciao, BinProlog, etc.), and the "delay technique" (applying negation as failure only *when* the variables of the negated goal become ground, which is sound but incomplete due to the possibility of floundering), which is present in Nu-Prolog, Gödel, and Prolog systems that implement delays (most of the above).

Of all the proposals, constructive negation [4, 5] is probably the most promising because it has been proven to be sound and complete, and its semantics is fully compatible with Prolog's. Constructive negation was, in fact, announced in early versions of the Eclipse Prolog compiler, but was removed from the latest releases. The reasons seem to be related to some technical problems with the use of coroutining (risk of floundering) and the management of constrained solutions.

The goal of this paper is to give an algorithmic description of constructive negation, i.e. explicitly stating the details needed for an implementation. We also intend to discuss the pragmatic ideas needed to provide a concrete and real implementation. Early results for a concrete implementation extending the Ciao Prolog compiler are presented. We assume some familiarity with constructive negation techniques and Chan's papers.

The remainder of the paper is organized as follows. Section 2 details our constructive negation algorithm. It explains how to obtain the $frontier$ of a goal (Section 2.1), how to prepare the goal for negation (Section 2.2) and, finally, how to negate the goal (Section 2.3). Section 3 discusses implementation issues: code expansion (Section 3.1), required disequality constraints (Section 3.2), optimizations (Section 3.3), examples (Section 4.1) and some experimental results (Section 4). Finally, we conclude and outline some future work.

2 Constructive Negation

Most of the papers addressing constructive negation deal with semantic aspects. In fact, only the original papers by Chan gave some hints about a possible implementation based on coroutining, but the technique was only outlined. When we tried to reconstruct this implementation we came across several problems, including the management of constrained answers and floundering (which appears to be the main reason why constructive negation was removed from recent versions of Eclipse). It is our belief that this problems cannot be easily and efficiently overcome. Therefore, we decided to design an implementation from scratch. One of our additional requirements is that we want to use a standard Prolog implementation (to be able to reuse thousands of existing Prolog lines and maintain their efficiency), so we will avoid implementation-level manipulations that would delay simple programs without negations.

We start with the definition of a frontier and how it can be managed to negate the respective formula.

2.1 Frontier

Firstly, we present Chan's definition of frontier (we actually owe the formal definition to Stuckey [15]).

Definition 1. Frontier
A frontier of a goal G is the disjunction of a finite set of nodes in the derivation tree such that every derivation of G is either finitely failed or passes through exactly one frontier node.

What is missing is a method to generate the frontier. So far we have used the simplest possible frontier: the frontier of depth 1 obtained by taking all the possible single SLD resolution steps. This can be done by a simple inspection of the clauses of the program[1]. Additionally, built-in based goals receive a special treatment (moving conjunctions into disjunctions, disjunctions into conjunction, eliminating double negations, etc.)

Definition 2. Depth-one frontier

- *If* $G \equiv (G_1; G_2)$ *then* $Frontier(G) \equiv Frontier(G_1) \vee Frontier(G_2)$.
- *If* $G \equiv (G_1, G_2)$ *then* $Frontier(G) \equiv Frontier(G_1) \wedge Frontier(G_2)$ *and then we have to apply DeMorgan's distributive property to retain the disjunction of conjunctions format.*
- *If* $G \equiv p(\overline{X})$ *and predicate* p/m *is defined by N clauses:*

  $p(\overline{X}^1) : -C'_1.$
  $p(\overline{X}^2) : -C'_2.$
  $\ldots$
  $p(\overline{X}^3) : -C'_N.$

  The frontier of the goal has the format: $Frontier(G) \equiv \{C_1 \vee C_2 \vee \ldots \vee C_N\}$, *where each* C_i *is the union of the conjunction of subgoals* C'_i *plus the equalities that are needed to unify the variables of* $\overline{X}$ *and the respective terms of* $\overline{X}^i$.

Consider, for instance, the following code:

```
odd(s(0)).
odd(s(s(X))) :- odd(X).
```

The frontier for the goal $odd(Y)$ is as follows:

$$Frontier(odd(Y)) = \{(Y = s(0)) \vee (Y = s(s(X)) \wedge odd(X))\}$$

To get the negation of G it suffices to negate the frontier formula. This is done by negating each component of the disjunction of all implied clauses (that form the frontier) and combining the results.

The solutions of $cneg(G)$ are the solutions of the combination (conjunction) of one solution of each of the N conjunctions C_i. Now we are going to explain how to negate a single conjunction C_i. This is done in two phases: *Preparation* and *Negation of the formula*.

[1] Nevertheless, we plan to improve the process by using abstract interpretation and detecting the degree of evaluation of a term that the execution will generate.

2.2 Preparation

Before negating a conjunction obtained from the frontier, we have to simplify, organize, and normalize this conjunction:

- **Simplification of the conjunction**. If one of the terms of C_i is trivially equivalent to $true$ (e.g. $X = X$), we can eliminate this term from C_i. Symmetrically,if one of the terms is trivially $fail$ (e.g. $X \neq X$), we can simplify $C_i \equiv fail$. The simplification phase can be carried out during the generation of frontier terms.
- **Organization of the conjunction**. Three groups are created containing the components of C_i, which are divided into equalities ($\overline{I}$), disequalities ($\overline{D}$), and other subgoals ($\overline{R}$). Then, we get $C_i \equiv \overline{I} \wedge \overline{D} \wedge \overline{R}$.
- **Normalization of the conjunction**. Let us classify the variables in the formula. The set of variables of the goal is called *GoalVars*. The set of free variables of $\overline{R}$ is called *RelVars*.
  - **Elimination of redundant variables and equalities**. If $I_i \equiv X = Y$, where $Y \notin GoalVars$, then we now have the formula $(I_1 \wedge \ldots \wedge I_{i-1} \wedge I_{i+1} \wedge \ldots \wedge I_{NI} \wedge \overline{D} \wedge \overline{R}\ )\sigma$, where $\sigma = \{Y/X\}$, i.e. the variable Y is substituted by X in the entire formula.
  - **Elimination of irrelevant disequalities**. *ImpVars* is the set of variables of *GoalVars* and the variables that appear in $\overline{I}$. The disequalities D_i that contain any variable that was neither in *ImpVars* nor in *RelVars* are irrelevant and should be eliminated.

2.3 Negation of the formula

It is not feasible, to get all solutions of C_i and to negate their disjunction because C_i can have an infinite number of solutions. So, we have to use the general constructive negation algorithm.

We consider that *ExpVars* is the set of variables of $\overline{R}$ that are not in *ImpVars*, i.e. *RelVars*, except the variables of $\overline{I}$ in the normalized formula.

First step: **Division of the formula**
C_i is divided into:

$$C_i \equiv \overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \overline{D}_{exp} \wedge \overline{R}_{exp}$$

where $\overline{D}_{exp}$ are the disequalities in $\overline{D}$ with variables in *ExpVars* and $\overline{D}_{imp}$ are the other disequalities, $\overline{R}_{exp}$ are the goals of $\overline{R}$ with variables in *ExpVars* and $\overline{R}_{imp}$ are the other goals, and $\overline{I}$ are the equalities.

Therefore, the constructive negation of the divided formula is:

$$\begin{aligned} \neg\, C_i \equiv\ & \neg\, \overline{I} \vee \\ & (\overline{I} \wedge \neg\, \overline{D}_{imp}) \vee \\ & (\overline{I} \wedge \overline{D}_{imp} \wedge \neg\, \overline{R}_{imp}) \vee \\ & (\overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \neg\, (\overline{D}_{exp} \wedge \overline{R}_{exp})) \end{aligned}$$

It is not possible to separate $\overline{D}_{exp}$ and $\overline{R}_{exp}$ because they contain free variables and they cannot be negated separately. The answers of the negations will be the answers of the negation of the equalities, the answers of the negation of the disequalities without free variables, the answers of the negation of the subgoals without free variables and the answers of the negation of the other subgoals of the conjunctions (the ones with free variables). Each of them will be obtained as follows:

Second step: **Negation of subformulas**

- **Negation of $\overline{I}$.** We have $\overline{I} \equiv I_1 \wedge \ldots \wedge I_{NI} \equiv$

$$\exists\, \overline{Z}_1\ X_1 = t_1 \wedge \ldots \wedge \exists\, \overline{Z}_{NI}\ X_{NI} = t_{NI}$$

  where $\overline{Z}_i$ are the variables of the equality I_i that are not included in *GoalVars* (i.e. that are not quantified and are therefore free variables). When we negate this conjunction of equalities we get the constraint

$$\underbrace{\forall\, \overline{Z}_1\ X_1 \neq t_1}_{\neg\, I_1} \vee \ldots \vee \underbrace{\forall\, \overline{Z}_{NI}\ X_{NI} \neq t_{NI}}_{\neg\, I_{NI}} \equiv \bigvee_{i=1}^{NI} \forall\, \overline{Z}_i X_i \neq t_i$$

  This constraint is the first answer of the negation of C_i that contains NI components.
- **Negation of $\overline{D}_{imp}$.** If we have $N_{D_{imp}}$ disequalities $\overline{D}_{imp} \equiv D_1 \wedge \ldots \wedge D_{N_{D_{imp}}}$ where $D_i \equiv \forall\ \exists\ \overline{Z}_i\ \overline{W}_i\ Y_i \neq s_i$ where Y_i is a variable of *ImpVars*, s_i is a term without variables in *ExpVars*, $\overline{W}_i$ are universally quantified variables that are neither in the equalities [2], nor in the other goals of $\overline{R}$ because otherwise $\overline{R}$ would be a disequality of $\overline{D}_{exp}$. Then we will get $N_{D_{imp}}$ new solutions with the format:

  $\overline{I} \wedge \neg\, D_1$
  $\overline{I} \wedge D_1 \wedge \neg\, D_2$
  $\ldots$
  $\overline{I} \wedge D_1 \wedge \ldots \wedge D_{N_{D_{imp}}-1} \wedge \neg\, D_{N_{D_{imp}}}$

  where $\neg\, D_i \equiv \exists\ \overline{W}_i\ Y_i = s_i$. The negation of a universal quantification turns into an existential quantification and the quantification of free variables of $\overline{Z}_i$ gets lost, because the variables are unified with the evaluation of the equalities of $\overline{I}$. Then, we will get $N_{D_{imp}}$ new answers.
- **Negation of $\overline{R}_{imp}$.** If we have $N_{R_{imp}}$ subgoals $\overline{R}_{imp} \equiv R_1 \wedge \ldots \wedge R_{N_{R_{imp}}}$. Then we will get new answers from each of the conjunctions:

  $\overline{I} \wedge \overline{D}_{imp} \wedge \neg\, R_1$
  $\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \neg\, R_2$
  $\ldots$
  $\overline{I} \wedge \overline{D}_{imp} \wedge R_1 \wedge \ldots \wedge R_{N_{R_{imp}}-1} \wedge \neg\, R_{N_{R_{imp}}}$

[2] There are, of course, no universally quantified variables in an equality

where $\neg\, R_i \equiv cneg(R_i)$. Constructive negation is again applied over R_i recursively using this operational semantics.

- **Negation of** $\overline{D}_{exp} \wedge \overline{R}_{exp}$. This conjunction cannot be disclosed because of the negation of $\exists\, \overline{V}_{exp}\, \overline{D}_{exp} \wedge \overline{R}_{exp}$, where $\overline{V}_{exp}$ gives universal quantifications: $\forall\, \overline{V}_{exp}\ cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$. The entire constructive negation algorithm must be applied again. Note that the new set *GoalVars* is the former set *ImpVars*. Variables of $\overline{V}_{exp}$ are considered as free variables. When solutions of $cneg(\overline{D}_{exp} \wedge \overline{R}_{exp})$ are obtained some can be rejected: solutions with equalities with variables in $\overline{V}_{exp}$. If there is a disequality with any of these variables, e.g. V, the variable will be universally quantified in the disequality. This is the way to negate the negation of a goal, but there is a detail that was not considered in former approaches and that is necessary to get a sound implementation: the existence of universally quantified variables in $\overline{D}_{exp} \wedge \overline{R}_{exp}$ by the iterative application of the method. So, what we are really negating is a subgoal of the form: $\exists\, \overline{V}_{exp}\ \overline{D}_{exp} \wedge \overline{R}_{exp}$. Here we will provide the last group of answers that come from:

$$\overline{I} \wedge \overline{D}_{imp} \wedge \overline{R}_{imp} \wedge \forall\, \overline{V}_{exp}\ \neg\, (\overline{D}_{exp} \wedge \overline{R}_{exp})$$

3 Implementation Issues

Having described the theoretical algorithm, including important details, we now discuss important aspects for a practical implementation, including how to compute the frontier and manage answer constraints.

3.1 Code Expansion

The first issue is how to get the frontier of a goal. It is possible to handle the code of clauses during the execution thanks to the Ciao package system [3], which allows the code to be expanded at run time. The expansion is implemented in the *cneg.pl* package which is included in the declaration of the module that is going to be expanded (i.e. where there are goals that are negations).

Note that a similar, but less efficient, behaviour can be emulated using metaprogramming facilities, available in most Prolog compilers.

3.2 Disequality constraints

An instrumental step for managing negation is to be able to handle disequalities between terms such as $t_1 \neq t_2$. The typical Prolog resources for handling these disequalities are limited to the built-in predicate `/== /2`, which needs both terms to be ground because it always succeeds in the presence of free variables. It is clear that a variable needs to be bound with a disequality to achieve a “constructive” behaviour. Moreover, when an equation $X = t(\overline{Y})$ is negated, the free variables in the equation must be universally quantified, unless affected by a more external quantification, i.e. $\forall\, \overline{Y}\ X \neq t(\overline{Y})$ is the correct negation. As we explained in [10], the inclusion of disequalities and constrained answers has a very low cost. It incorporates negative normal form constraints instead of bindings and the decomposition step can produce disjunctions.

A Prolog predicate `=/= /2` [10] has been defined, used to check disequalities, similarly to explicit unification (=). Each constraint is a disjunction of conjunctions of disequalities. When a universal quantification is used in a disequality (e.g., $\forall Y\ X \neq c(Y)$), the new constructor `fA/1` is used (e.g., `X / c(fA(Y)))`.

3.3 Optimizing the algorithm and the implementation

Our constructive negation algorithm and the implementation techniques admit some additional optimizations that can improve the runtime behaviour of the system. Basically, the optimizations rely on the compact representation of information, as well as the early detection of successful or failing branches.

Compact information. In our system, negative information is represented quite compactly, providing fewer solutions from the negation of $\overline{I}$. The advantage is twofold. On the one hand constraints contain more information and failing branches can be detected earlier (i.e. the search space could be smaller). On the other hand, if we ask for all solutions using backtracking, we are cutting the search tree by offering all the solutions together in a single answer. For example, we can offer a simple answer for the negation of a predicate p (the code for p is skipped):

```
?- cneg(p(X,Y,Z,W)).

(X=/=0, Y=/=s(Z)) ; (X=/=Y) ; (X=/=Z) ;
(X=/=W) ; (X=/=s(0), Z=/=0) ? ;
no
```

(which is equivalent to $(X \neq 0 \wedge Y \neq s(Z)) \vee X \neq Y \vee X \neq Z \vee X \neq W \vee X \neq s(0) \vee Z \neq 0$), instead of returning six answers upon backtracking:

```
?- cneg(p(X,Y,Z,W)).

X=/=0, Y=/=s(Z) ? ;
X=/=Y ? ;
X=/=Z ? ;
X=/=W ? ;
X=/=s(0) ? ;
Z=/=0 ? ;
no
```

Pruning subgoals. The frontiers generation search tree can be cut with a double action over the ground subgoals: removing the subgoals whose failure we are able to detect early on, and simplifying the subgoals that can be reduced to true. Suppose we have a predicate $p/2$ defined as

```
p(X,Y):- greater(X,Y),
         q(X,Y,Z),
         r(Z).
```

where $q/3$ and $r/1$ are predicates defined by several clauses with a complex computation. To negate the goal $p(s(0),s(s(0)))$, its frontier is computed:

$$Frontier(p(s(0),s(s(0)))) \equiv$$

$$X = s(0) \wedge Y = s(s(0)) \wedge greater(X,Y) \wedge q(X,Y,Z) \wedge r(Z) \equiv$$

$$greater(s(0),s(s(0))) \wedge q(s(0),s(s(0)),Z) \wedge r(Z) \equiv$$

$$fail \wedge q(s(0),s(s(0)),Z) \wedge r(Z) \equiv$$

$$fail$$

The next step is to expand the code of the subgoals of the frontier to the combination (disjunction) of the code of all their clauses, and the result will be a very complicated and hard to check frontier. However, the process is optimized by evaluating ground terms. In this case, $greater(s(0),s(s(0))$ fails and, therefore, it is not necessary to continue with the generation of the frontier, because the result is reduced to fail (i.e. the negation of $p(s(0),s(s(0)))$ will be trivially true). The opposite example is a simplification:

$$Frontier(p(s(s(0)),s(0))) \equiv$$

$$X = s(s(0)) \wedge Y = s(0) \wedge greater(X,Y) \wedge q(X,Y,Z) \wedge r(Z) \equiv$$

$$greater(s(s(0)),s(0)) \wedge q(s(s(0)),s(0),Z) \wedge r(Z) \equiv$$

$$true \wedge q(s(s(0)),s(0),Z) \wedge r(Z) \equiv$$

$$q(s(s(0)),s(0),Z) \wedge r(Z)$$

Constraint simplification. During the whole process for negating a goal,the frontier variables are constrained. In cases where the constraints are satisfiable, they can be eliminated and where the constraints can be reduced to fail, the evaluation can be stopped with result *true*.

We focus on the negative information of a normal form constraint F:

$$F \equiv \bigvee_i \bigwedge_j \forall\, \overline{Z}^i_j\ (Y^i_j \neq s^i_j)$$

Firstly, the Prenex form [13] can be obtained by extracting the universal variables with different names to the head of the formula, applying logic rules:

$$F \equiv \forall \overline{x} \bigvee_i \bigwedge_j (Y^i_j \neq s^i_j)$$

and using the distributive property:

$$F \equiv \forall \overline{x} \bigwedge_k \bigvee_l (Y^k_l \neq s^k_l)$$

The formula can be separated into subformulas that are simple disjunctions of disequalities :

$$F \equiv \bigwedge_k \forall \overline{x} \bigvee_l (Y_l^k \neq s_l^k) \equiv F_1 \wedge \ldots \wedge F_n$$

Each single formula F_k can be evaluated. The first step will be to substitute the existentially quantified variables (variables that do not belong to $\overline{x}$) by Skolem constants s_j^i that will keep the equivalence without losing generality:

$$F_k \equiv \forall \overline{x} \bigvee_l (Y_l^k \neq s_l^k) \equiv \forall \overline{x} \bigvee_l (Y_{Skl}^k \neq s_{Skl}^k)$$

Then it can be transformed into:

$$F_k \equiv \neg \exists\, \overline{x} \neg (\bigvee_l (Y_{Skl}^k \neq s_{Skl}^k)) \equiv \neg Fe_K$$

The meaning of F_k is the negation of the meaning of Fe_k;

$$Fe_k \equiv \exists\, \overline{x} \neg (\bigvee_l (Y_{Skl}^k \neq s_{Skl}^k))$$

Solving the negations, the result is obtained through simple unifications of the variables of $\overline{x}$:

$$Fe_k \equiv \exists\, \overline{x} \bigwedge \neg (Y_{Skl}^k \neq s_{Skl}^k) \equiv \exists\, \overline{x} \bigwedge (Y_{Skl}^k = s_{Skl}^k)$$

Therefore, we get the truth value of F_k from the negation of the value of Fe_k and, finally, the value of F is the conjunction of the values of all F_k. If F succeeds, then the constraint is removed because it is redundant and we continue with the negation process. If it fails, then the negation directly succeeds.

4 Experimental results

Our prototype is a simple library that is added to the set of libraries of Ciao Prolog. Indeed, it is easy to port the library to other Prolog compilers. The only requirement is that attributed variables should be available.

This section reports some experimental results from our prototype implementation. First of all, we show the behaviour of the implementation in some simple examples.

4.1 Examples

The interesting side of this implementation is that it returns constructive results from a negative question. Let us start with a simple example involving predicate $boole/1$.

```
boole(0).
boole(1).
```

```
?- cneg(boole(X)).
X=/=1, X=/=0 ? ;
no
```

Another simple example obtained from [15] gives us the following answers:

```
p(a,b,c).                      ?- proof1(X,Y,Z).
p(b,a,c).
p(c,a,b).                      Z = c,
                               X=/=b, X=/=a ? ;
proof1(X,Y,Z):-
    X =/= a,                   Z = c,
    Z = c,                     Y=/=a, X=/=a ? ;
    cneg(p(X,Y,Z)).
                               no
```

[15] contains another example showing how a constructive answer ($\forall T\ X \neq s(T)$) is provided for the negation of an undefined goal in Prolog:

```
p(X):- X = s(T), q(T).         ?- r(X).

q(T):- q(T).                   X=/=s(fA(_A)) ?

r(X):- cneg(p(X)).             yes
```

Notice that if we would ask for a second answer, then it will loop according to the Prolog resolution. An example with an infinite number of solutions is more interesting.

```
                          ?- cneg(positive(X)).

                          X=/=s(fA(_A)), X=/=0 ? ;
positive(0).              X = s(_A),
positive(s(X)):-          (_A=/=s(fA(_B)), _A=/=0) ? ;
        positive(X).      X = s(s(_A)),
                          (_A=/=s(fA(_B)), _A=/=0) ? ;
                          X = s(s(s(_A))),
                          (_A=/=s(fA(_B)), _A=/=0) ?
                          yes
```

4.2 Implementation measures

We have firstly measured the execution times in milliseconds for the above examples when using negation as failure ($naf/1$) and constructive negation ($cneg/1$). A '-' in a cell means that negation as failure is not applicable. All measurements were made using Ciao Prolog[3] 1.5 on a Pentium II at 350 MHz. The results are shown in Table 1. We have added a first column with the runtime of the evaluation of the positive goal that is negated in the other columns and a last column with the ratio that measures the speedup of the *naf* technique w.r.t. constructive negation.

Using **naf** instead of **cneg** results in small ratios around 1.06 on average for ground calls with few recursive calls. So, the possible slow-down for constructive negation is not so high as we might expect for these examples. Furthermore, the results are rather

[3] The negation system is coded as a library module ("package" [3]), which includes the respective syntactic and semantic extensions (i.e. Ciao's attributed variables). Such extensions apply locally within each module which uses this negation library.

goals	Goal	naf(Goal)	cneg(Goal)	ratio
boole(1)	2049	2099	2069	0.98
boole(8)	2070	2170	2590	1.19
positive(s(s(s(s(s(s(0))))))	2079	1600	2159	1.3
positive(s(s(s(s(s(0))))))	2079	2139	2060	0.96
greater(s(s(s(0))),s(0))	2110	2099	2100	1.00
greater(s(0),s(s(s(0))))	2119	2129	2089	0.98
average				1.06
positive(500000)	2930	2949	41929	14.21
positive(1000000)	3820	3689	81840	22.18
greater(500000,500000)	3200	3339	22370	7.70
average				14.69
boole(X)	2080	-	3109	
positive(X)	2020	-	7189	
greater(s(s(s(0))),X)	2099	-	6990	
greater(X,Y)	7040	-	7519	
queens(s(s(0)),Qs)	6939	-	9119	

Table 1. Runtime comparation

similar. But the same goals with data that involve many recursive calls yield ratios near 14.69 on average w.r.t **naf**, increasing exponentially with the number of recursive calls. There are, of course, many goals that cannot be negated using the *naf* technique and that are solved using constructive negation.

5 Conclusion and Future Work

After running some preliminary experiments with the constructive negation technique following Chan's description, we realized that the algorithm needed some additional explanations and modifications.

Having given a detailed specification of algorithm in a detailed way we proceed to provide a real, complete and consistent implementation. The result, we have reported are very encouraging, because we have proved that it is possible to extend Prolog with a constructive negation module relatively inexpensively. Nevertheless, it is quite important to address possible optimizations, and we are working to improve the efficiency of the implementation. These include a more accurate selection of the frontier based on the demanded form of argument in the vein of [9]). Other future work is to incorporate our algorithm at the WAM machine level.

In any case, we will probably not be able to provide an efficient enough implementation of constructive negation, because the algorithm is inherently inefficient. This is why we do not intend to use it either for all cases of negation or for negating goals directly.

Our goal is to design and implement a practical negation operator and incorporate it into a Prolog compiler. In [10, 11] we systematically studied what we understood to be the most interesting existing proposals: negation as failure (*naf*) [6], use of delays to apply *naf* securely [12], intensional negation [1, 2], and constructive negation [4, 5, 7, 14, 15]. As none of them can satisfy our requirements of completeness and efficiency, we propose to use a combination of these techniques, where the information from static program analyzers could be used to reduce the cost of selecting techniques [11]. So, in many cases, we avoid the inefficiency of constructive negation. However, we still need it because it is the only method that is sound and complete for all kinds of goal. For example, looking at the goals in Table 1, the strategy will obtain all ground negation using the *naf* technique and would only use constructive negation for the goals with variables where it is impossible to use *naf*.

We are testing the implementation and trying to improve the code, and our intention is to include it in the next version of Ciao Prolog [4].

References

1. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. Intensional negation of logic programs. *LNCS*, 250:96–110, 1987.
2. R. Barbuti, D. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *JLP*, 8(3):201–228, 1990.
3. D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.
4. D. Chan. Constructive negation based on the complete database. In *Proc. Int. Conference on LP'88*, pages 111–125. The MIT Press, 1988.
5. D. Chan. An extension of constructive negation and its application in coroutining. In *Proc. NACLP'89*, pages 477–493. The MIT Press, 1989.
6. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, New York, NY, 1978. Plenum Press.
7. W. Drabent. What is a failure? An approach to constructive negation. *Acta Informatica.*, 33:27–59, 1995.
8. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.
9. J. J. Moreno-Navarro. Extending constructive negation for partial functions in lazy narrowing-based languages. *ELP*, 1996.
10. S. Muñoz and J. J. Moreno-Navarro. How to incorporate negation in a prolog compiler. In E. Pontelli and V. Santos Costa, editors, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer-Verlag.
11. S. Muñoz, J. J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, number 2250 in LNAI, pages 485–494, La Habana (Cuba), 2001. LPAR 2001.
12. L. Naish. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.
13. J. R. Shoenfield. *Mathematical Logic*. Association for Symbolic Logic, 1967.
14. P. Stuckey. Constructive negation for constraint logic programming. In *Proc. IEEE Symp. on Logic in Computer Science*, volume 660. IEEE Comp. Soc. Press, 1991.
15. P. Stuckey. Negation and constraint logic programming. In *Information and Computation*, volume 118(1), pages 12–33, 1995.

[4] http://www.clip.dia.fi.upm.es/Software