The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

ICLP'06 Workshop

CICLOPS 2006: Colloquium on Implementation of Constraint LOgic Programming Systems

August 21st, 2006

Proceedings

Editors:

H-F. Guo and E. Pontelli

Preface

The last years have witnessed continuous progress in the technology available both for academic and commercial computing environments. Examples include more processor performance, increased memory capacity and bandwidth, faster networking technology, and operating system support for cluster computing. These improvements, combined with recent advances in compilation and implementation technologies, are causing high-level languages to be regarded as good candidates for programming complex, real world applications. Techniques aiming at achieving flexibility in the language design make powerful extensions easier to implement; on the other hand, implementations which reach good performance in terms of speed and memory consumption make declarative languages and systems amenable to develop non-trivial applications.

Logic Programming and Constraint Programming, in particular, seem to offer one of the best options, as they couple a high level of abstraction and a declarative nature with an extreme flexibility in the design of their implementations and extensions and of their execution model. This adaptability is key to, for example, the implicit exploitation of alternative execution strategies tailored for different applications (e.g., for domain-specific languages) without unnecessarily jeopardizing efficiency.

This workshop continues a tradition of successful workshops on Implementations of Logic Programming Systems, previously held with 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 (ParImp) held in Port Jefferson (1997), Manchester (1998), Las Cruces (1999), and London (2000), and more recently the Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS) in Paphos (Cyprus, 2001), Copenhagen (2002), Mumbai (2003), Saint-Malo (France, 2004), and Sitges (Spain, 2005), and the CoLogNet Workshops on Implementation Technology for Computational Logic Systems held in Madrid (2002), Pisa (2003) and Saint-Malo (France, 2004).

The workshop aims at discussing and exchanging experience on the design, implementation, and optimization of logic and constraint (logic) programming systems, or systems intimately related to logic as a means to express computations.

August 2006

Hai-Feng Guo & Enrico Pontelli
Workshop Chairs
CICLOPS'06

Organization

CICLOPS'06 is organized by the Departments of Computer Science at the University of Nebraska at Omaha and New Mexico State University, in cooperation with the 2006 Federated Logic Conference (FLoC) and the Association for Logic Programming.

Organizing Committee

Conference Chairs:	Hai-Feng Guo (University of Nebraska at Omaha, USA) Enrico Pontelli (New Mexico State University, USA)

Program Commitee

Manuel Carro	Polytechnic University of Madrid (Spain)
Bart Demoen	KUL Leuven (Belgium)
Michel Ferreira	University of Porto (Portugal)
Gopal Gupta	University of Texas at Dallas (USA)
Vitor Santos Costa	Federal University of Rio de Janeiro (Brasil)
Tom Schrijvers	KUL Leuven (Belgium)
Christian Schulte	KTH Royal Institute of Technology (Sweden)
Neng-Fa Zhou	City University of New York (USA)

Table of Contents

Invited Talk: AR (Action Rules): The Language, Implementation, and Applications 1
Neng-Fa Zhou

Efficient Support for Incomplete and Complete Tables in the YapTab Tabling System 2
Ricardo Rocha

When Tabling Does Not Work 18
Remko Tronçon, Bart Demoen, Gerda Janssens

Embedding Solution Preferences via Transformation 32
Hai-Feng Guo and Miao Liu

Towards Region-based Memory Management for Mercury Programs 47
Quan Phan and Gerda Janssens

Delay and Events in the TOAM and the WAM 63
Bart Demoen and Phuong-Lan Nguyen

On Applying Deductive Databases to Inductive Logic Programming 80
Tiago Soares, Michel Ferreira, Ricardo Rocha, Nuno Fonseca

DBTAB: a Relational Storage Model for the YapTab Tabling System 95
Pedro Costa, Ricardo Rocha, Michel Ferreira

Linear Logic: Foundations, Applications and Implementations 110
Lukas Chrpa

AR (Action Rules): The Language, Implementation, and Applications (Invited Talk)

Neng-Fa Zhou[1]

CUNY Brooklyn College and Graduate Center,
zhou@sci.brooklyn.cuny.edu

Abstract. AR is a rule-based event-driven language designed for programming interactions needed in such applications as propagation-based constraint solvers and graphical user interfaces. AR is a result of evolution from delay constructs in early logic programming systems and it supports events on domain variables available in several constraint programming systems for programming constraint propagators. AR is compiled into a spaghetti-stack machine which facilitates fast switching of subgoals between suspended and active states. This architecture has been shown to be vital for the high performance of the finite-domain constraint solver and the CHR (Constraint Handling Rules) compiler in B-Prolog. This tutorial is based on the papers [1,2,3,4].

References

1. Neng-Fa Zhou. Programming Finite-Domain Constraint Propagators in Action Rules. To appear in *Theory and Practice of Logic Programming.* 2006.
2. Neng-Fa Zhou, Mark.Wallace, and Peter J. Stuckey. The dom Event and its Use in Implementing Constraint Propagators. Technical Report, CUNY Compuer Science, 2006.
3. Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. Translating Constraint Handling Rules into Action Rules. *CHR'06*, 2006.
4. Neng-Fa Zhou. A Constraint-based Graphics Library for B-Prolog. *Software - Practice and Experience*, Vol. 33, No.13, pp.1199-1216, 2003.

Efficient Support for Incomplete and Complete Tables in the YapTab Tabling System

Ricardo Rocha

DCC-FC & LIACC
University of Porto, Portugal
ricroc@ncc.up.pt

Abstract. Most of the recent proposals in tabling technology were designed as a means to improve some practical deficiencies of current tabling execution models. The discussion we address in this paper was also motivated by practical deficiencies we encountered, in particular, when dealing with incomplete and complete tables. Incomplete tables became a problem when, as a result of a pruning operation, the computational state of a tabled subgoal is removed from the execution stacks before being completed. On the other hand, complete tables became a problem when the system runs out of memory. To handle incomplete tables, we propose an approach that avoids re-computation when the already stored answers are enough to evaluate a repeated call. To handle complete tables, we propose a memory management strategy that automatically recovers space from the tables when the system runs out of memory. To validate our proposals, we have implemented them in the YapTab tabling system as an elegant extension of the original design.

1 Introduction

Resolution strategies based on tabling [1,2] are able to reduce the search space, avoid looping, and have better termination properties than traditional Prolog models based on SLD resolution [3]. As a result, in the past years several alternative tabling models have been proposed [4,5,6,7,8,9] and implemented in systems like XSB, Yap, B-Prolog, ALS-Prolog and Mercury.

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for current subgoals in an appropriate data space, called the *table space*. Whenever a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses. The added power of tabling has proved its viability in application areas such as Knowledge Based Systems [10,11], Program Analysis [12], and Inductive Logic Programming [13].

More recently, the increasing interest in tabling technology led to further developments and proposals that improve some practical deficiencies of current tabling execution models. In [14], Sagonas and Stuckey proposed a mechanism, named *just enough tabling*, that offers the capability to arbitrarily suspend and resume a tabled evaluation without requiring full re-computation. In [15], Saha

and Ramakrishnan proposed an incremental evaluation algorithm for maintaining the freshness of tables that avoids recomputing the full set of answers when the program changes upon addition or deletion of facts/rules. In [16], Rocha *et al.* proposed the ability to support dynamic mixed-strategy evaluation of the two most successful tabling scheduling strategies, batched and local scheduling.

All these recent proposals were designed as a means to improve the performance of particular applications in key aspects of tabled evaluation like re-computation and scheduling. The discussion we address in this work was also motivated by our recent attempt [13] of applying tabling to Inductive Logic Programming (ILP). ILP applications are very interesting for tabling because they have huge search spaces and do a lot of re-computation. Moreover, we found that they are an excellent case study to improve some practical deficiencies of current tabling execution models. In particular, in this paper we focus on the table space and how to efficiently handle incomplete and complete tables.

A table is said to be *complete* when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program for the subgoal call associated with the table. Otherwise, it is said to be *incomplete*. A table for a tabled subgoal is thus marked as complete when, during evaluation, it is determined that all possible resolutions for the subgoal have been made and, therefore, no more answers can be found.

Incomplete tables became a problem when, as a result of a pruning operation, the computational state of a tabled subgoal is removed from the execution stacks before being completed. We may have found several answers but not the complete set. Thus, when a repeated call appears, we cannot simply load answers from an incomplete table, because we may loose part of the computation. The usual approach implemented in most tabling systems is to throw away the already found answers and restart the evaluation from the beginning when a repeated call appears. In this paper, we propose a more aggressive approach and, by default, we keep incomplete tables for pruned subgoals. Then, later, when a repeated call appears, we start by consuming the available answers from the incomplete table, and only if we exhaust all such answers, we restart the evaluation from the beginning. The idea is to avoid any re-computation when the already stored answers are enough to evaluate a repeated call.

On the other hand, complete tables can also be a problem, but when the system runs out of memory space. For such cases we need to compromise efficiency and throw away some of the tables in order to recover space and let the computation continue. Memory handling is a serious problem when we use tabling for applications that store large answers and/or a huge number of answers. The common control implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables. In this paper, we propose a more robust approach, a memory management strategy, based on a *least recently used* algorithm, that automatically recovers space from the tables when the system runs out of memory.

To validate our proposals, we have implemented them in the YapTab tabling system as an elegant extension of the original design [5]. To the best of our knowl-

edge, YapTab is the first tabling system that implements support for incomplete and complete tables as discussed above. Results using the April ILP system [17] showed very substantial performance gains and a substantial increase of the size of the problems that can be solved by combining ILP with tabling. Despite the fact that we use ILP as the motivation for this work, the problems, proposals and results that we discuss next are not restricted to ILP applications and can be generalised and applied to any other application.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts and discuss the motivation for our work. Next, we present our proposals and describe the issues involved in providing engine support for integrating them in the YapTab tabling system. We then present some experimental results and outline some conclusions.

2 Background and Motivation

To discuss the motivation for our work, we start by introducing some basic concepts about tabling and ILP and then we address the practical deficiencies encountered when combining them.

2.1 Basic Tabling Definitions

Tabling is about storing answers for subgoals so that they can be reused when a repeated call appears. The nodes in a tabled evaluation are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals. Tabling based models have four main types of operations for definite programs:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table, and allocates a new generator node.
2. The *new answer* operation verifies whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.
3. The *answer resolution* operation verifies whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no unconsumed answers are available, it *suspends* the current computation and schedules a backtracking node to continue the execution.
4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. It executes when backtracking to a generator node and all of its clauses have been tried. If the subgoal is completely evaluated, the operation marks the corresponding table entry as complete and reclaims stack space. Otherwise, control moves to a consumer with unconsumed answers.

We could delay completion until the very end of the 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* [18] to detect whether a generator node has been fully exploited and, if so, to recover space for all its consumers. Moreover, if we call a repeated subgoal that is already completed, then we can avoid consumer node allocation and perform instead what is called a *completed table optimization* [19]. 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.

2.2 Inductive Logic Programming

The fundamental goal of an ILP system is to find a consistent and complete theory (logic program), from a set of examples and prior knowledge, the *background knowledge*, that explains all given positive examples, while being consistent with the given negative examples. Since it is not usually obvious which set of hypotheses should be picked as the theory, an ILP system must traverse the *hypotheses space* searching for a set of hypotheses (clauses) with the desired properties.

Computing the coverage of a hypothesis requires, in general, running positives and negatives examples against the clause. For instance, to evaluate if the hypothesis '`theory(X):- a1(X),a2(X,Y).`' covers the example `theory(p1)`, the system executes the goal `once(a1(p1),a2(p1,Y))`. The `once/1` predicate is a primitive that prunes over the search space preventing the unnecessary search for further answers. It is defined in Prolog as '`once(Goal):- call(Goal),!.`'. Note that the ILP system is only interested in evaluating the coverage of the hypothesis, and not in finding answers for the goal.

Now assume that the previous hypothesis obtains a *good coverage*, that is, the number of positive examples covered by it is high and the number of negative example is low. Then, it is quite possible that the system will use it to generate more specific hypotheses such as '`theory(X):- a1(X),a2(X,Y),a3(Y).`'. If the same example, `theory(p1)`, is then evaluated against this new hypothesis, goal `once(a1(p1),a2(p1,Y),a3(Y))`, part of the computation will be repeated. For data-sets with a large number of examples, we can arbitrarily do a lot of recomputation. Tabling technology is thus an excellent candidate to significantly reduce the execution time for these kind of problems.

2.3 Tabling and Inductive Logic Programming

Consider now that we declare predicate `a2/2` as tabled. Coverage computation with tabled evaluation works fine when examples are not covered by hypotheses. In such cases, all the tabled subgoals in a clause are completed. For instance, when evaluating the goal `once(a1(p1),a2(p1,Y),a3(Y))`, if the subgoal `a3(Y)` never succeeds then, by backtracking, `a2(p1,Y)` will be completely evaluated. On the other hand, tabled evaluation can be a problem when examples are successfully covered by hypotheses. For example, if `once(a1(p1),a2(p1,Y),a3(Y))` eventually succeeds, then the `once/1` primitive will reclaim space by pruning the goal at hand. However, as `a2(p1,Y)` may still succeed with other answers for

`Y`, its table entry cannot be marked as complete. Thus, when a repeated call to `a2(p1,Y)` appears, we cannot simply load answers from its incomplete table, because we may loose part of the computation. A question then arises: how can we make tabling worthwhile in an environment that potentially generates so many incomplete tables?

In previous work [13], we first studied this problem by taking advantage of YapTab's functionality that allows it to combine different scheduling strategies within the same evaluation [16]. Our results showed that best performance can be achieved when we evaluate some subgoals using *batched scheduling* and others using *local scheduling*. Batched scheduling is the default strategy, it schedules the program clauses in a depth-first manner as does the WAM. This strategy favors forward execution, when new answers are found the evaluation automatically propagates the answer to solve the goal at hand. Local scheduling is an alternative strategy that tries to *force* completion before returning answers. The key idea is that whenever new answers are found, they are added to the table space, as usual, but execution fails. Answers are only returned when all program clauses for the subgoal at hand were resolved.

At first, local scheduling seems more attractive because it avoids the pruning problem mentioned above. When the `once/1` primitive prunes the search space, the tables are already completed. On the other hand, if the cost of fully generating the complete set of answers is very expensive, then the ILP system may not always benefit from it. It can happen that, after completing a subgoal, the subgoal is not called again or when called it succeeds just by using the initial answers, thus, making it useless to compute beforehand the full set of answers. Another major problem with mixed-strategy evaluation, is that, from the programmer's point of view, it is very difficult to define beforehand the subgoals to table using one or another strategy. The approach we propose in this work can be seen as a compromise between the efficiency of batched scheduling and the effectiveness of local scheduling. We want to favor forward execution in order to quickly succeed with the coverage evaluation of the hypotheses, but we also want to be able to reuse the already found answers in order to avoid re-computation.

When applying tabling to ILP, we can also explore the fact that an important characteristic of ILP systems is that they generate candidate hypotheses which have many similarities among them. Usually, these similarities tend to correspond to common *prefixes* (conjunction of subgoals) among the candidate hypotheses. Thus, if we are able to table these conjunction of subgoals, we only need to compute them once. This strategy can be recursively applied as the system generates more specific hypotheses. This idea is similar to the *query packs* technique proposed by Blockeel *et al.* [20].

However, to recursively table conjunction of subgoals, we need to store a large number of tables, and thus, we may increase the table memory usage arbitrarily and quickly run out of memory [13]. Therefore, at some point, we need to compromise efficiency and throw away some of the tables in order to recover space and let the computation continue. A first approach is to let the programmer dynamically control the deletion of the tables. However, this can be hard

to implement and difficult to decide what are the potentially useless tables that should be deleted. In order to allow useful deletion without compromising efficiency, in this work, we propose a more robust approach, a memory management strategy based on a *least recently used* replacement algorithm that automatically recovers space from the tables when the system runs out of memory.

We next describe how we extended the YapTab tabling system to provide engine support for handling incomplete and complete tables as discussed above.

3 Handling Incomplete Tables

This section describes our proposal to handle incomplete tables. The main goal of our proposal is to avoid re-computation when the answers in an incomplete table are enough to evaluate a repeated call. To support that, we thus keep incomplete tables for pruned subgoals. Then, when a repeated call to a pruned subgoal appears, we start by consuming the available answers from its incomplete table, and only if we exhaust all such answers, we restart the evaluation from the beginning. Later, if the subgoal is pruned again, then the same process is repeated until eventually the subgoal is completely evaluated. We next describe how we extended the YapTab tabling system to support incomplete tables.

3.1 Implementation Details

In YapTab, tables are implemented using *tries* as proposed in [19]. An important data structure in the table space is the *subgoal frame*. For each different tabled subgoal call, a different subgoal frame is used to store information about the subgoal. In particular, part of that information includes a pointer to where answers are stored, the `SgFr_answers` field, and a flag indicating the state of the subgoal, the `SgFr_state` field (see Fig. 1 for details).

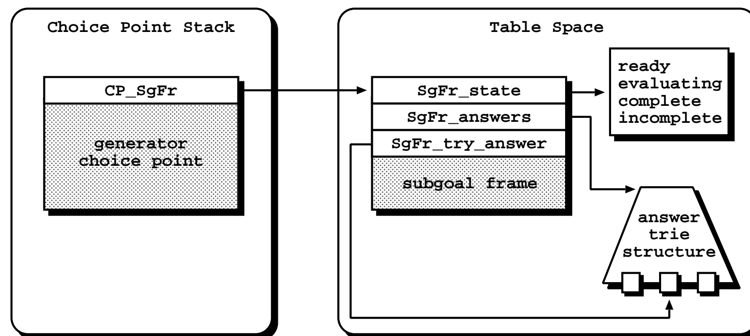


Fig. 1. Generator choice points and subgoal frames in YapTab

During evaluation, a subgoal frame can be in one of the following states: *ready*, i.e., without a corresponding generator in the choice point stack; *evaluating*, i.e., with a generator being evaluated; or *complete*, i.e., with the generator

no longer present but with the subgoal fully evaluated. At the engine level, generator nodes are implemented as WAM choice points extended with two extra fields. One of these fields, the `CP_SgFr` field, points to the associated subgoal frame in the table space.

To support incomplete tables, we have introduced two minor changes to the subgoal frame data structure. First, a new *incomplete* state, marks the subgoals whose corresponding generators were pruned from the execution stacks. Second, when we are consuming answers from an incomplete table as a result of a repeated call to a previously pruned subgoal, a new `SgFr_try_answer` field marks the currently loaded answer (similarly to what consumer nodes have). As an optimization, if a subgoal has no answers when its generator is pruned, then we can avoid marking its table as incomplete. Instead, we can mark it as ready and, when a repeated call appears, proceed as if it was the first call.

Handling incomplete tables also required minor changes to the tabled subgoal call operation. Figure 2 shows how we extended the `tabled_subgoal_call()` instruction to deal with incomplete tables.

```
tabled_subgoal_call(subgoal SG) {
  sg_fr = search_table_space(SG)     // sg_fr is the subgoal frame for SG
  if (SgFr_state(sg_fr) == ready) {
    gen_cp = store_generator_node(sg_fr)
    SgFr_state(sg_fr) = evaluating
    CP_AP(gen_cp) = failure_continuation_instruction()   // second clause
    goto next_instruction()
  } else if (SgFr_state(sg_fr) == evaluating) {
    cons_cp = store_consumer_node(sg_fr)
    goto answer_resolution(cons_cp)            // start consuming answers
  } else if (SgFr_state(sg_fr) == complete) {
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == incomplete) {      // new block of code
    gen_cp = store_generator_node(sg_fr)
    SgFr_state(sg_fr) = evaluating
    first = get_first_answer(sg_fr)
    load_answer_from_trie(first)
    SgFr_try_answer(sg_fr) = first      // mark the current loaded answer
    CP_AP(gen_cp) = table_try_answer                     // new instruction
    goto continuation_instruction()
  }
}
```

Fig. 2. Pseudo-code for `tabled_subgoal_call()`

The new block of code that deals with incomplete tables is similar to the block that deals with first calls to tabled subgoals (ready state flag). It also stores a generator node, but instead of using the program clauses to evaluate the subgoal call, as usual, it starts by loading the first available answer from the incomplete table. The subgoal's `SgFr_try_answer` field is made to point to this first answer.

A second difference is that the failure continuation pointer of the generator choice point, the `CP_AP` field, is now updated to a special `table_try_answer` instruction.

The `table_try_answer` instruction implements a variant of the answer resolution operation (see section 2.1). Figure 3 shows the pseudo-code for it. Initially, the `table_try_answer` instruction checks if there are more answers to be consumed, and if so, it loads the next one and updates the `SgFr_try_answer` field. When this is not the case, all available answers have been already consumed. Thus, we need to restart the computation from the beginning. The program counter is made to point to the first clause corresponding to the subgoal call at hand and the failure continuation pointer of the generator is updated to the second clause. At this point, the evaluation is in the same computational state as if we had executed a first call to the tabled subgoal call operation. The difference is that the table space for our subgoal already stores some answers.

```
table_try_answer(generator GEN) {
  sg_fr = CP_SgFr(GEN)
  last = SgFr_try_answer(sg_fr)             // get the last loaded answer
  next = get_next_answer(last)
  if (next) {                                  // answers still available
    load_answer_from_trie(next)
    SgFr_try_answer(sg_fr) = next     // update the current loaded answer
    goto continuation_instruction()
  } else {                // restart the evaluation from the first clause
    PREG = get_compiled_code(sg_fr)        // PREG is the program counter
    CP_AP(GEN) = failure_continuation_instruction()      // second clause
    goto next_instruction()
  }
}
```

Fig. 3. Pseudo-code for `table_try_answer()`

We should remark that the use of generator nodes to implement the calls to incomplete tables is strictly necessary to maintain unalterable all the remaining data structures and algorithms of the tabling engine. Note that, at the engine level, these calls are again the first representation of the subgoal in the execution stacks because the previous representation has been pruned.

3.2 Discussion

Let us consider again the previous ILP example and the evaluation of the goal `once(a1(p1),a2(p1,Y),a3(Y))` with predicate `a2/2` declared as tabled. Consider also that, after a long computation for `a2(p1,Y)`, we found three answers: `Y=y1`, `Y=y2`, and `Y=y3`, and that `a3(Y)` only succeeds for `Y=y3`. Primitive `once/1` then prunes the goal at hand and `a2(p1,Y)` is marked as incomplete. Now assume that, later, the ILP system calls again `a2(p1,Y)` when evaluating a different goal, for example, `once(a2(p1,Y),a4(Y))`. If `a4(Y)` succeeds with one of the previously found answers: `Y=y1`, `Y=y2`, or `Y=y3`, then no evaluation will be required for subgoal `a2(p1,Y)`. This is the typical case where we can profit from

having incomplete tables. The gain in the execution time is proportional to the cost of evaluating the subgoal from the beginning until generating the proper answer.

On the other hand, if `a4(Y)` does not succeed with any of the previously found answers, then `a2(p1,Y)` will be reevaluated as a first call. Therefore, for such a case, we do not take any advantage of having maintained the incomplete table. This means that all the answers stored in the table, `Y=y1`, `Y=y2` and `Y=y3`, will be generated again. However, as these answers are repeated, `a4(Y)` will not be called again for them. The evaluation will fail until a non-repeated answer is eventually found. Thus, the computation time required to evaluate `once(a2(p1,Y),a4(Y))` for these answers, either with or without the incomplete table, is then equivalent. Therefore, we may not benefit from having maintained the incomplete table, but we do not pay any cost either.

Our proposal is closer to the spirit of the *just enough tabling (JET)* proposal of Sagonas and Stuckey [14]. In a nutshell, the JET proposal offers the capability to arbitrarily suspend and resume a tabled evaluation without requiring any re-computation. The basic idea is that JET copies the execution stacks corresponding to pruned subgoals to an auxiliary area in order to be able to resume them later when a repeated call appears. The authors argue that the cost of JET is linear in the number of choice points which are pruned. However, to the best of our knowledge, no practical implementation of JET was yet been done.

Compared to JET, our approach does not require an auxiliary data space, does not require any complex dependencies to maintain information about pruned subgoals, and does not introduce any overhead in the pruning process. We thus believe that the simplicity of our approach can produce comparable results to JET when applied to real applications like ILP applications.

4 Handling Complete Tables

This section describes our proposal to handle complete tables when the system runs out of memory. We propose a memory management strategy that automatically recovers space from the least recently used tables. Note that this proposal is completely orthogonal to the previous one, that is, we can support both independently but we can also support both simultaneously. In what follows, we will consider the case where YapTab also includes support for incomplete tables as described in the previous section. Therefore and despite the fact that here we are focusing the case of complete tables, we will include in the spirit of our proposal the case of incomplete tables too.

4.1 Implementation Details

In YapTab, each subgoal call is represented by a different subgoal frame in the table space. Besides this representation, a subgoal can also be represented in the execution stacks. First calls to tabled subgoals or calls to pruned subgoals

are represented by generator nodes; repeated calls to tabled subgoals are represented by consumer nodes; and calls to completed subgoals are represented by interior nodes that execute compiled code directly from the answer trie structure associated with the completed subgoal. A subgoal is said to be *active* if it is represented in the execution stacks. Otherwise, it is said to be *inactive*. Inactive subgoals are thus only represented in the table space.

A subgoal can also be in one of the following states: ready, evaluating, complete or incomplete. The ready and incomplete states correspond to situations where the subgoal is inactive, while the evaluating state corresponds to a situation where the subgoal is active. The complete state is a special case because it can correspond to both active and inactive situations. In order to be able to distinguish these two situations, we introduced a new state named *complete-active*. We use the complete-active state to mark the completed subgoals that are also active in the execution stacks, while the previous complete state is used to mark the completed subgoals that are only represented in the table space. With this simple extension, we can now use the `SgFr_state` field of the subgoal frames to decide if a subgoal is currently active or inactive.

Knowing what subgoals are active or inactive is important when the system runs out of memory. Obviously, active subgoals cannot be removed from the table space because otherwise we may loose part of the computation or produce errors. Therefore, when the system runs out of memory, we should try to recover space from the inactive subgoals. Figure 4 shows how we extended the YapTab system to handle inactive subgoals.

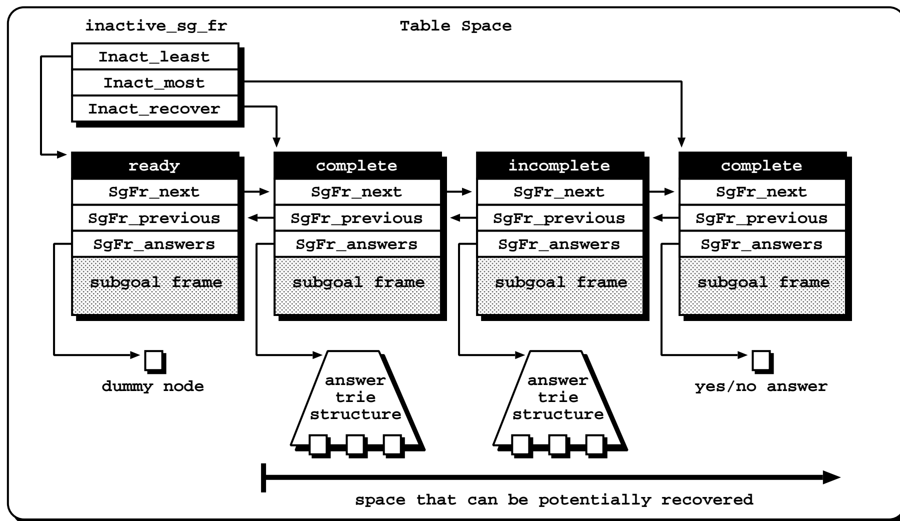


Fig. 4. Handling inactive subgoals in YapTab

Subgoal frames corresponding to inactive subgoals are stored in a double linked list that is accessible by a new data structure named `inactive_sg_fr`. The `Inact_least` and `Inact_most` fields point respectively to the least and most recently inactive subgoal frames. A third field, `Inact_recover`, points to the

next inactive subgoal frame from where space can be potentially recovered. Two subgoal frame fields, `SgFr_next` and `SgFr_previous`, link the list. Space from inactive subgoals is recovered as presented next in Fig. 5.

```
recover_space(structure data type STR_TYPE) {
  // STR_TYPE is the data type that we failed to allocate space for
  sg_fr = Inact_recover(inactive_sg_fr)
  do {
    if (sg_fr == NULL)                                    // end of list
      return
    if (get_first_answer(sg_fr)) {          // subgoal frame with answers
      free_answer_trie_structure(sg_fr)                   // recover space
      SgFr_state(sg_fr) = ready                  // reset the frame state
    }
    sg_fr = SgFr_next(sg_fr)
  } while (no_space_available_for(STR_TYPE))
  Inact_recover(inactive_sg_fr) = sg_fr            // update recover field
}
```

Fig. 5. Pseudo-code for `recover_space()`

The `recover_space()` procedure is called when the system fails to allocate memory space for a specific data type, the `STR_TYPE` argument. It starts from the subgoal frame pointed by the `Inact_recover` field and then it uses the `SgFr_next` field to navigate in the list of inactive subgoals until at least a page of memory be recovered for the given data type. YapTab uses a page-based memory allocation scheme where each page only stores data structures of the same type, and thus, to start using a memory page to allocate a different data structure we first need to completely deallocate all the previous data structures from the page.

When recovering space, we only consider the subgoals that store at least one answer (completed subgoals with a yes/no answer are kept unalterable) and for these we only recover space from their answer trie structures. Through experimentation we found that, for a large number of applications, the space required by all the other table data structures is insignificant when compared with the space required by the answer trie structures (usually more than 99% of the total space). Therefore, only sporadically, we are able to recover space from the non-answer related data structures. We thus argue that the potential benefit of recovering space from these structures does not compensate its cost.

During evaluation, an inactive subgoal can be made active again. This occurs when we execute a repeated call to an inactive subgoal. For such cases, we thus need to remove the corresponding subgoal frame from the `inactive_sg_fr` list. On the other hand, when a subgoal turns inactive, its subgoal frame is inserted in the `inactive_sg_fr` list as the most recently inactive frame. A subgoal turns inactive when it executes completion, it is pruned or it fails from an interior node that was executing compiled code from the answer trie structure.

This latter case can be complicated because we can have several interior nodes executing compiled code from the same answer trie. Only when the computation fails from the last (older) interior node should the corresponding subgoal be made inactive. To correctly implement that we use the trail stack. The call that first executes code for a completed subgoal changes the subgoal's state to complete-active and stores in the trail stack the reference to the subgoal frame. Further calls for the same subgoal (cases where the subgoal's state is now complete-active) are handled as before. Figure 6 shows how we extended the `tabled_subgoal_call()` instruction to support this.

```
tabled_subgoal_call(subgoal SG) {
  sg_fr = search_table_space(SG)     // sg_fr is the subgoal frame for SG
  if (SgFr_state(sg_fr) == ready) {
    remove_from_inactive_list(sg_fr)                              // new
    ...
  } else if (SgFr_state(sg_fr) == evaluating) {
    ...
  } else if (SgFr_state(sg_fr) == complete) {
    remove_from_inactive_list(sg_fr)                              // new
    SgFr_state(sg_fr) = complete-active                           // new
    trail(sg_fr)                                                  // new
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == complete-active) {        // new state
    goto SgFr_answers(sg_fr)       // execute compiled code from the trie
  } else if (SgFr_state(sg_fr) == incomplete) {
    remove_from_inactive_list(sg_fr)                              // new
    ...
  }
}
```

Fig. 6. Extended pseudo-code for `tabled_subgoal_call()`

When later backtracking occurs, we use the reference in the trail stack to correctly insert the subgoal in the `inactive_sg_fr` list. This use of the trail stack does not introduce any overhead because the YapTab engine already uses the trail to store information beyond the normal variable trailing (to control dynamic predicates, multi-assignment variables and frozen segments).

4.2 Discussion

A common control implemented in most tabling systems, YapTab included, is to have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables. With our approach, the programmer can still force the deletion of particular tables, but can also rely on the effectiveness of the memory management algorithm to completely avoid the problem of deciding what potentially useless tables should be deleted.

However, we can still increase the table memory space arbitrarily. This can happen if the space required by the set of active subgoals exceeds the available memory space and we are not able to recover any space from the set of inactive subgoals. A possible solution for this problem is to store data externally using, for example, a relational database system. We are already studying how this can be done, that is, how we can partially move tables to database storage and efficiently load them back to the tabling engine. This idea can also be applied to inactive subgoals and, in particular, we can eventually use our memory management algorithm, not to decide what tables to delete but, to decide what tables to move to the database.

5 Experimental Results

To evaluate the impact of our proposals, we ran the April ILP system [17] with YapTab. The environment for our experiments was a Pentium M 1600MHz processor with 1 GByte of main memory and running the Linux kernel 2.6.11.

We first experimented our support to handle incomplete tables and, for that, we used a well-known ILP data-set, the *Mutagenesis* data-set, with two different configurations that we named *Mutagen1* and *Mutagen2*. The main difference between the configurations is that the hypotheses space is searched differently. Table 1 shows the running times, in seconds, for *Mutagen1* and *Mutagen2* using four different approaches to evaluate the predicates in the background knowledge: **(i)** without tabling; **(ii)** using local scheduling; **(iii)** using batched scheduling; and **(iv)** using batched scheduling with support for incomplete tables. The running times include the time to run the whole ILP system. During evaluation, *Mutagen1* and *Mutagen2* call respectively 1479 and 1461 different tabled subgoals and, for batched scheduling, both end with 76 incomplete tables.

Tabling Mode	*Mutagen1*	*Mutagen2*
Without tabling	> 1 day	> 1 day
Local scheduling	153.9	143.3
Batched scheduling	278.2	137.9
Batched scheduling with incomplete tables	122.9	117.6

Table 1. Running times with and without support for incomplete tables

Our results show that, by combining batched scheduling with incomplete tables, we can further speedup the execution for these kind of problems. Batched scheduling allows us to favor forward execution and incomplete tables allows us to avoid re-computation. However, for some subgoals, local scheduling can be better than batched scheduling with incomplete tables. We can benefit from local scheduling when the cost of fully generating the complete set of answers is less than the cost of evaluating the subgoal several times as a result of several pruning operations. Better results are thus still possible if we use YapTab's flexibility that allows to intermix batched with local scheduling within the same evaluation. However, from the programmer point of view, it is very difficult to define the subgoals to table using one or another strategy. We thus argue that

our combination of batched scheduling with incomplete tables is an excellent (and perhaps the best) compromise between simplicity and good performance.

We next show how we used another well-known ILP data-set, the *Carcinogenesis* data-set, to experiment with our second proposal. From our previous work on tabling conjunctions of subgoals, we selected one of the hypotheses that allocates more memory when computing its coverage against the set of examples in the *Carcinogenesis* data-set. That hypothesis is defined by a prefix that represents the conjunction of 5 tabled subgoals with a total of 20 arguments. Table 2 shows the running times in seconds (or *m.o.* for memory overflow) for computing its coverage with four different table limit sizes: 576, 384, 192 and 128 MBytes (the table limit size is defined statically when the system starts). In parentheses, it shows the number of executions of the `recover_space()` procedure.

Tabling Mode	576MB	384MB	192MB	128MB
Local scheduling	15.2	15.9(95)	16.9(902)	*m.o.*(893)
Batched scheduling	11.4	12.6(62)	14.1(523)	*m.o.*(557)
Batched scheduling with incomplete tables	11.1	12.3(91)	13.9(833)	*m.o.*(833)

Table 2. Running times with different table limit sizes

Through experimentation, we found that this computation requires a total table space of 576 MBytes if not recovering any space, and a minimum of 160 MBytes if using our recovering mechanism (for Pentium-based architectures, YapTab allocates memory in segments of 32 MBytes). The results obtained with this particular example show that batched scheduling with incomplete tables is again the best approach. The results also suggest that our recovering mechanism is quite effective in performing its task (for a memory reduction of 66% in table space it introduces an average overhead between 10% and 20% in the execution time). The impact of our proposal in the execution time depends, in general, on the size of the table space and on the specificity of the application being evaluated, i.e., on the number of times it may call subgoals whose tables were previously deleted by the recovering procedure.

6 Conclusions

In this paper, we have discussed some practical deficiencies of current tabling systems when dealing with incomplete and complete tables. Incomplete tables became a problem when, as a result of a pruning operation, the computational state of a tabled subgoal is removed from the execution stacks before being completed. On the other hand, complete tables became a problem when the system runs out of memory space.

To handle incomplete tables, we proposed the ability to avoid re-computation by keeping incomplete tables for pruned subgoals. The typical case where we can profit from having incomplete tables is, thus, when the already stored answers are enough to evaluate repeated calls. When this is not the case, we cannot benefit from it but, on the other hand, we do not pay any cost too. To handle complete tables, we proposed a memory management strategy that automatically recovers

space from inactive tables when the system runs out of memory. Both proposals have been implemented in the YapTab tabling system with minor changes to the original design. Preliminaries results using the April ILP system showed very substantial performance gains and a substantial increase of the size of the problems that can be solved by combining ILP with tabling.

Acknowledgments

We are very thankful to Nuno Fonseca for his support with the April ILP System. This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
3. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
4. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
5. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
6. Demoen, B., Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling. Future Generation Computer Systems **16** (2000) 809–830
7. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
8. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. Journal of Functional and Logic Programming **2001** (2001)
9. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
10. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
11. Yang, G., Kifer, M.: Flora: Implementing an Efficient Dood System using a Tabling Logic Engine. In: Computational Logic. Number 1861 in LNCS, Springer-Verlag (2000) 1078–1093
12. Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S., Dong, Y., Du, X., Roychoudhury, A., Venkatakrishnan, V.: XMC: A Logic-Programming-Based Verification Toolset. In: International Conference on Computer Aided Verification. Number 1855 in LNCS, Springer-Verlag (2000) 576–580

13. Rocha, R., Fonseca, N., Costa, V.S.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning. Number 3720 in LNAI, Springer-Verlag (2005) 707–714
14. Sagonas, K., Stuckey, P.: Just Enough Tabling. In: ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM (2004) 78–89
15. Saha, D., Ramakrishnan, C.R.: Incremental Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 235–249
16. Rocha, R., Silva, F., Costa, V.S.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264
17. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)
18. Chen, W., Swift, T., Warren, D.S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. Journal of Logic Programming **24** (1995) 161–199
19. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
20. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16** (2002) 135–166

When Tabling Does Not Work

Remko Tronçon*, Bart Demoen, and Gerda Janssens

Katholieke Universiteit Leuven, Dept. of Computer Science,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{remko,bmd,gerda}@cs.kuleuven.be

Abstract. Tabled execution has been successfully applied in various domains such as program analysis, model checking, parsing, ... A recent target of tabling is the optimization of Inductive Logic Programming. Due to the iterative nature of ILP algorithms, queries evaluated by these algorithms typically show a lot of similarity. To avoid repeated execution of identical parts of queries, answers to the queries can be tabled and reused in later steps of the algorithm, thus avoiding redundancy. In this paper, we make a qualitative evaluation of this application of tabling, and compare it with query packs, a special execution mechanism for sets of similar queries. We show that the memory overhead introduced by tabling not only scales poorly in this context, but that query pack execution also avoids more predicate calls for more complex problems.

1 Introduction

By remembering and reusing answers to goals, *tabled execution* (or *tabling*) can speed up execution of logic programs, and make predicates terminate where they previously did not. Many applications benefit from tabled execution, including program analysis, model checking, parsing, ... Recently, there has been research into the applicability of tabling for Inductive Logic Programming data mining. ILP algorithms try to discover interesting patterns in large sets of data by iteratively generating queries and evaluating them on the data (represented as sets of logic programs). Because queries are generated by extending queries from previous iterations of the algorithm, the queries evaluated are typically very similar. By applying tabling to the query evaluation process, repeated execution of identical parts of queries can be reduced to retrieving their solution from memory [7]. On the other hand, *query packs* [3] have been designed to overcome the redundancy in execution of similar queries as well. By grouping queries in a special type of disjunction, execution of identical goals is shared over different queries, thus only executing them once.

In this paper, we analyze the application of tabling in the context of Inductive Logic Programming. We evaluate three approaches that employ tabling to optimize ILP execution, and compare them with query packs. Because no system we know of provides both support for query packs and tabling, our comparison

* Supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.)

is purely qualitative, based on the number of goal calls occurring.

The organization of the paper is as follows: In Section 2, we give a brief introduction to ILP, the specific context in which this work takes place. Section 3 presents a motivating example illustrating some of the advantages and disadvantages of different query execution alternatives, tabled and untabled. In Section 4, we describe a slightly adapted version of the prefix and query tabling approaches described in [7]. Section 5 introduces once-tabling, a lighter version of query tabling that limits the total memory overhead. In Section 6, we make a comparison between the various tabling approaches and query pack execution. Finally, we conclude in Section 7.

2 Background: ILP Query Execution

We start by sketching the context in which this work is situated, namely the execution of queries in Inductive Logic Programming (ILP). The goal of ILP is to find a theory that best explains a large set of data (or examples). In the ILP setting at hand, each example is a logic program, and the theory is represented as a set of logical queries. The ILP algorithm finds good queries by using generate-and-test: generated queries are run on sets of examples; based on the failure or success of these queries, only the ones with the 'best' results are kept and are extended (refined) by adding literals. The number of literals that are added is dictated by the lookahead setting. For example, an ILP system without lookahead set could extend a query

```
?- a, b.
```

into the queries

```
?- a, b, c.
?- a, b, f.
```

where only one literal is added to the original query. With a lookahead setting of 1, the query can be extended with 2 literals:

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

The criterium to select which of the generated queries are best depends on the ILP algorithm: for example, in the case of classification, the information gain can be used as a criterion, whereas in the case of regression, the reduction of variance is often used. The extended queries are in turn tested on a set of examples, and this process continues until a satisfactory query (or set of queries) describing the examples has been found.

Queries executed by ILP algorithms generally consist of a prefix and a refinement part. The prefix is a query from a previous iteration that was selected by

the algorithm for further extension, and the refinement itself is a new conjunction. The set of evaluated queries is therefore partitioned into sets of queries with an identical prefix. This means that during the execution of these queries over an example, the same answers for the prefix are computed over and over again. To avoid this recomputation, it makes sense to remember the computed answers to prefixes, and reuse them when subsequent queries with the same prefix are executed. We call this approach *prefix tabling*. Because the answers stored by prefix tabling are only used within the same iteration of the ILP algorithm, they can be removed from memory in the next iteration.

Another consequence of the incremental nature described above is that, since the prefix of a query is itself a query from the previous iterations, each prefix has been executed before (and either yielded an answer or failed). Remembering the results of queries can therefore avoid execution of prefixes in the next iterations. We call this approach *query tabling*. Because the answers to queries depend on answers from previous iterations of the ILP algorithm, all answers from the previous iterations need to be stored in memory as long as there are queries depending on them. Therefore, contrary to prefix tabling, the extra memory used for remembering the answers cannot be freed after every iteration. Both approaches are presented in [7].

A different approach to avoiding redundancy in the execution of similar queries are *query packs* [3]. Instead of separately executing a set of similar queries on the data set, the set of queries is transformed into a special kind of disjunction, called a query pack. For example, the set of queries

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

is transformed into the query

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

by applying left factoring on the initial set of queries. This new query shares the execution of the common prefix `a,b` over all queries, and additionally shares the execution of `c` in the first two queries. However, because in ILP only the success of a query on an example is important (not the actual solution), the normal Prolog disjunction might still cause too much backtracking. So, for efficiency reasons the `';'/2` is given a different procedural behavior in query packs: it cuts away branches from the disjunction as soon as they succeed, thus avoiding it is executed again after success. We denote this alternative version of the disjunction as $;_p$. Since each query pack is run on a large set of examples, a query pack is in practice first compiled, and the compiled code is executed on the examples. This compiled code makes use of special WAM instructions for the query pack execution mechanism. More details can be found in [3]. Notice that, contrary to query tabling and prefix tabling, query packs do not introduce memory overhead.

Iteration 1
1 a(X), b(X,Y)
2 m(X), n(X,X)

Iteration 2
3 (a(X),b(X,Y)), c(Y,Z),d(Z)
4 (a(X),b(X,Y)), c(Y,Z),e(Z)

Iteration 3
5 (a(X),b(X,Y),c(Y,Z),d(Z)), f(Z)
6 (a(X),b(X,Y),c(Y,Z),e(Z)), g(Z)

Example 1		
a(1).	c(1,1).	e(2).
a(2).	c(1,2).	f(2).
b(2,1).	d(2).	g(2).

Example 2			
a(1).	c(1,1).	d(3).	f(15).
b(1,1).	c(1,2).	d(15).	g(15).
	⋮	e(3).	
	c(1,15).	e(15).	

Fig. 1. Example query trace.

Example	Experiment	Query								
		1	**2**	**1+2**	**3**	**4**	**3+4**	**5**	**6**	**5+6**
1	No Optimization	4	1	5	8	8	16	9	9	18
	Prefix Tabling	4	1	5	8	4	12	9	9	18
	Query Tabling	4	1	5	4	4	8	1	1	2
	Query Packs			5			10			12
2	No Optimization	2	1	3	8	8	16	34	34	68
	Prefix Tabling	2	1	3	8	6	14	34	34	68
	Query Tabling	2	1	3	6	6	12	26	26	52
	Query Packs			3			11			51

Table 1. Number of calls (including backtracking) for different execution mechanisms.

3 Motivating Example

We illustrate the advantages of prefix tabling, query tabling, and query packs using the example from Figure 1. This example shows the queries executed in 3 iterations of an ILP algorithm, and two data set examples on which the queries are executed. Table 1 shows for every query how many goals were called or backtracked to.

First, let us consider the execution of the queries on the first example. Executing the prefix of the first query of the second iteration, which is a refinement of a query from the first iteration, results in a call and a redo for `a`, and 2 calls to `b` before reaching the solution {X=2,Y=1}. The remainder of the query execution consists of a call and a redo for `c` and 2 calls for `d`, resulting in a final total of 6 calls and 2 redos for the first query. However, in the second query, the

same prefix is executed. By remembering the solution to the prefix, execution of the calls to `a` and `b` can be replaced by fetching the solution for the prefix from memory, and the only calls remaining are the call and redo to `c` and both calls to `e`, resulting in a total of 3 calls and one redo for the second query (instead of 6 calls and 2 redos without reusing the answers from the prefix). Additionally, remembering the answers to the queries in the first iteration avoids execution of the prefixes in the second iteration altogether, resulting in a total of 6 calls and 2 redos for the second iteration (instead of 9 calls and 3 redos when only the answers for the prefix are remembered).

The third iteration from Figure 1 consists of refinements of both queries from the previous iteration. Consequently, the prefixes of the queries are not identical, and prefix tabling will therefore not be able to reuse previously computed solutions. Remembering the answers to the full queries from the previous iteration remedies this, and saves execution of every prefixes.

Transforming the queries from the second iteration of Figure 1 in a query pack results in the following pack:

```
?- a(X),b(X,Y),c(Y,Z),
          (   d(Z)
          ;p  e(Z)).
```

When executing this pack on the first example, the prefix of the queries is executed only once over the whole iteration, as is the case with prefix tabling. In the third iteration, the following query pack is executed:

```
?- a(X),b(X,Y),c(Y,Z),
          (   d(Z),f(Z)
          ;p  e(Z),g(Z)).
```

In this case, query pack execution outperforms prefix tabling: because both prefixes are not identical, prefix tabling is unable to reuse any computed answer, and both prefixes are executed separately for every query; however, the execution of `a`, `b` and `c` is shared in the query pack, leading to fewer calls for the third iteration.

Although query tabling outperforms query packs on the first example, this is not always the case. This is illustrated by the results of running the queries on the second example from Figure 1. In the second iteration, query pack execution has the advantage that the execution of `c` in the refinement is shared between both queries, whereas it is executed separately with the query tabling approach. The same goes for the prefix of the queries in the third iteration, where the execution of `a`, `b`, and `c` is shared by query pack execution, whereas with query tabling only the answers to (`a(X)`,`b(X,Y)`) are reused during the execution of both queries.

One can easily see that query pack execution always performs at least as well as prefix tabling if it comes to the total number of calls to goals: both approaches

execute identical prefixes only once, but query packs can additionally exploit the similarity of non-identical prefixes. For query tabling and query packs, none of the approaches always outperforms the other. Remembering the answers of queries across iterations can avoid executing prefixes, whereas query packs always need to execute the prefix at least once. However, by exploiting similarity in refinements and prefixes, query packs can also provide an advantage over query tabling in the case where queries are refined with more than one literal at a time (i.e. with lookahead enabled).

4 Prefix and Query Tabling

In this section, we describe the approach taken to perform both prefix tabling and query tabling. This approach conforms to the approach taken in [7], only without tabling the calls to separate goals themselves. The tabling of separate goals is left out because it is independent of the actual execution mechanism of queries (i.e. query tabling or query packs).

Queries which are to be evaluated are of the following form:

$$?-\ \mathit{Prefix}(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{1}$$

where *Prefix* and *Refinement* are both conjunctions, and $\overline{x}$ and $\overline{y}$ are the sets of variables occurring in them respectively. In the first iteration of the ILP algorithm, *Prefix* is always empty (`true`); in the next iterations, *Prefix* is a query from the previous iteration, and *Refinement* is either a goal or a conjunction of goals, added to the query in the current iteration.

For every query of the form (1) to be evaluated, do the following:

1. If no such predicate has been created yet, create a predicate
$$\mathtt{P}_i(\overline{x}) :-\ \mathit{Prefix}(\overline{x}). \tag{2}$$
(where i is a unique identifier for *Prefix*), and table the answers for $\mathtt{P}_i(\overline{x})$.
2. Transform the query into
$$?-\ \mathtt{P}_i(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{3}$$
3. Evaluate the transformed query.

For example, suppose the query

$$?-\ \mathtt{a(X,Y),b(Y,Z)}$$

is refined into the following set of queries:

$$?-\ \mathtt{(a(X,Y),b(Y,Z)),c(Z,U).}$$
$$?-\ \mathtt{(a(X,Y),b(Y,Z)),d(Z,U),e(U).}$$

(the prefix is put between brackets for notational purposes). The tabling mechanism transforms these queries into

$$?-\ \mathtt{P_{a_b}(X,Y,Z),c(Z,U)} \tag{4}$$
$$?-\ \mathtt{P_{a_b}(X,Y,Z),d(Z,U),e(U)} \tag{5}$$

and creates a new tabled predicate

$$\mathtt{P_{a_b}(X,Y,Z)}\ :-\ \mathtt{a(X,Y),b(Y,Z)}.$$

Executing the transformed version of these queries re-uses previously computed answers of the prefix, thus avoiding redundancy in execution. After having evaluated the queries, the newly created predicates and their tables can be cleared, thus keeping the extra memory usage local to the iteration. We call this approach *prefix tabling*.

Prefix tabling can be extended further to yield full *query tabling*. This is achieved by transforming the query (3) further into

$$?-\ \mathtt{P}_j(\overline{x}\cup\overline{y})$$

where $\mathtt{P}_j(\overline{x}\cup\overline{y})$ is a new tabled predicate, defined as follows:

$$\mathtt{P}_j(\overline{x}\cup\overline{y})\ :-\ \mathtt{P}_i(\overline{x}), \mathit{Refinement}(\overline{y}) \tag{6}$$

Due to the prefix transformation in the next iteration, one of the tabled full queries is re-used when executing the prefix of the refined queries. For example, the transformed queries (4) and (5) are transformed further into

$$?-\ \mathtt{P_{a_b_c}(X,Y,Z,U)}$$
$$?-\ \mathtt{P_{a_b_d_e}(X,Y,Z,U)}$$

with the new tabled predicates

$$\mathtt{P_{a_b_c}(X,Y,Z,U)}\ :-\ \mathtt{P_{a_b}(X,Y,Z),c(Z,U)}.$$
$$\mathtt{P_{a_b_d_e}(X,Y,Z,U)}\ :-\ \mathtt{P_{a_b}(X,Y,Z),d(Z,U),e(U)}.$$

Suppose the first query is refined in the next iteration into

$$?-\ (\mathtt{P_{a_b}(X,Y,Z),c(Z,U)}),\mathtt{d(U,V)}$$

The prefix transformation transforms this query into

$$?-\ \mathtt{P_{a_b_c}(X,Y,Z,U),d(U,V)}$$

which is exactly the tabled predicate created for the full query in the previous iteration.

Iteration 1
1 a(X), b(X,Y)

Iteration 2
2 (a(X),b(X,Y)), c(Y).

Example 1
a(1). b(2,1). c(1).
a(2). b(2,2). c(2).

Example 2
a(1). b(2,2). c(3).
a(2). b(2,3).

Fig. 2. Example query trace.

5 Once Tabling

We observed that during the execution of queries, many times only the first solution for a query is used when executing the refinement of the query in a later iteration. It is therefore worth investigating the performance of a weaker alternative to the query tabling approach from Section 4, where the weaker version only stores one answer for every succeeded query. We call this approach *once-tabling*.

We start by illustrating the intuition behind once-tabling using the example from Figure 2. After the first iteration of Figure 2 finished, the query succeeded with answers {X=2,Y=1} and {X=2,Y=2} for the two examples respectively. Instead of executing the query from the second iteration on the first example, we first transform the query to

$$?-\ (\mathtt{X}=2, \mathtt{Y}=1\ ;\ \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{b}(\mathtt{X},\mathtt{Y})), \mathtt{c}(\mathtt{Y}).$$

This transformed query reuses the previously computed answer to its prefix by immediately binding the variables to their solution, and using the original prefix if this solution makes the refinement fail. Executing this transformed query on the second example indeed avoids the execution of the prefix, as the query succeeds immediately when calling d after binding Y to 1. For the second example, the corresponding transformed query is:

$$?-\ (\mathtt{X}=2, \mathtt{Y}=2\ ;\ \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{b}(\mathtt{X},\mathtt{Y})), \mathtt{c}(\mathtt{Y}).$$

In this case, however, the previously computed solution {X=2,Y=2} of the prefix does not lead to a solution of the refinement. The prefix therefore has to be executed as normal.

Once-tabling can be implemented as a simple query transformation. Suppose again that a query is of the form (1):

$$?-\ \mathit{Prefix}(\overline{x}), \mathit{Refinement}(\overline{y}).$$

We transform this query into

$$?-\ (\mathtt{load_solution}(i,\overline{x})\ ; \mathit{Prefix}(\overline{x})),\ \mathit{Refinement}(\overline{y}), \mathtt{save_solution}(j,\overline{x}\cup\overline{y}).$$

where i is a unique identifier for *Prefix*, j a unique identifier for the whole query. The predicates `load_solution` and `save_solution` store for a given key the variable bindings of their second argument. The solutions to queries are stored separately for each example. Executing this transformed query first retrieves the previously computed solution for the prefix, and then executes the refinement. If this solution fails to satisfy the refinement, the original prefix is executed.

A first advantage of the once-tabling approach is that the extra memory required for storing answers to queries is limited to at most one solution per query. The second advantage is that it is relatively easy to implement this approach in any system, without necessarily having to resort to tabled execution. The disadvantage of this approach is that the prefix sometimes needs to be re-executed, although this extra overhead is compensated for if the majority of the queries succeed using the first solution of their prefix.

To be able to make a fair comparison between once tabling and the query tabling approach from Section 4, we implemented once tabling similarly to the implementation of query tabling, i.e. by using tabled execution. For every query of the form (1), we create the following predicate:

$$\mathtt{P}_j(\overline{x} \cup \overline{y}) \ :- \ \mathtt{once}(((\mathtt{P}_i(\overline{x}); \mathit{Prefix}(\overline{x})), \mathit{Refinement}(\overline{y})))$$

where i and j are unique identifiers for *Prefix* and *(Prefix,Refinement)* respectively. Similar to the predicate (6) created for query tabling, this predicate is also tabled. The query itself is then transformed into

$$?- \ \mathtt{P}_j(\overline{x} \cup \overline{y}).$$

With this transformation, only the first answer to every query will be stored in memory.

For example, a query from the first iteration (without prefix)

$$?- \ \mathtt{a(X,Y),b(Y,Z)}$$

is transformed into

$$?- \ \mathtt{P_{a_b}(X,Y,Z)}.$$

with predicate $\mathtt{P_{a_b}}$ tabled, and defined as

$$\mathtt{P_{a_b}(X,Y,Z)} \ :- \ \mathtt{once((a(X,Y),b(Y,Z)))}.$$

A refinement of this query

$$?- \ \mathtt{(a(X,Y),b(Y,Z)),c(Z,U)}$$

is transformed into

$$?- \ \mathtt{P_{a_b_c}(X,Y,Z,U)}$$

with predicate $\mathtt{P_{a_b_c}}$ also tabled, and defined as

$$\mathtt{P_{a_b_c}(X,Y,Z,U)} \ :- \ \mathtt{once(((P_{a_b}(X,Y,Z);a(X,Y),b(Y,Z)),c(Z,U)))}.$$

6 Evaluation

The main goal of our evaluation is to see whether the tabling approaches described in Sections 4 and 5 provide an advantage over the query packs approach (at the cost of extra memory for storing the answers). Query packs have previously been implemented in the core of hipP [6], the engine underlying the ACE data mining system [1]. However, the query and prefix tabling approaches require support for tabled execution, which hipP does not provide. We therefore built a prototype for the tabling approaches, based on the implementation of [7] in YAP [4]. On the other hand, YAP does not support query pack execution, making it impossible to compare the query pack and the tabling approaches based on timing results. We therefore estimate the potential benefit of these techniques by measuring the total number of goals that are actually called, and comparing these to each other. We do this by running an ILP algorithm from the ACE system once, recording all the queries executed by ACE in a file (which we call a *trace*). Using these traces, we can simulate the execution step of the ILP algorithm without needing the algorithm itself: by running a trace through a simple program (which we call a *trace simulators*) that executes every query on the corresponding examples, the same queries can be executed than the ones that are run by executing the algorithm, independent of the system used (hipP or YAP). Notice that the number of calls is not a strict measure of the time needed to execute a query: the time to execute a call can vary widely, and the time needed to look up tabled results is also not taken into account in an evaluation based purely on the number of calls. It does, however, provide a rough indication of the performance that can be expected.

In our experiments, we used two ILP algorithms: Tilde [2], a decision tree learner, and Warmr [5], a frequent pattern discovery algorithm. Traces from both algorithms were recorded for different lookahead settings on both the Mutagenesis [9] and Carcinogenesis [8] data sets. These traces were adorned with calls to predicates recording the number of calls, and the resulting traces were fed to three different trace simulators: one that executes the queries from a trace in their original form, one that first applies the tabling transformations on the queries before executing the queries, and one that executes query packs. YAP provides two different modes for tabling predicates: *local* tabling computes the complete table of a called predicate before continuing execution, while in *batched* tabling the table is constructed by need. Batched tabling therefore in theory performs less calls to predicates than local tabling. However, due to the way YAP handles the combination of tabling with the cut at the end of each query, answers of a query sometimes need to be recomputed when the table needs to be completed further.

The total number of goal calls for the different Tilde runs are shown in Tables 2 and 4. As expected, query packs outperform prefix tabling in all experiments, and query tabling performs better than query packs in the settings without lookahead. However, with lookahead enabled, query packs start compensating the prefix recomputation cost by taking advantage of similarity in the prefix and refinements, and as such outperform query tabling with a higher

Experiment	Lookahead		
	0	1	2
No Optimization	390715	1347047	169003392
Prefix Tabling (Local)	296134	937022	9287876
Prefix Tabling (Batched)	296283	937120	9288178
Query Tabling (Local)	105560	542372	4013514
Query Tabling (Batched)	52115	443086	3667033
Once Tabling	103237	777236	69336452
Query Packs	184267	444453	2407452

Table 2. Total number of goal calls for running Tilde on Mutagenesis.

Experiment	Lookahead		
	0	1	2
Prefix Tabling (Local)	37 kB	37 kB	36 kB
Prefix Tabling (Batched)	37 kB	37 kB	36 kB
Query Tabling (Local)	10 MB	15 MB	92 MB
Query Tabling (Batched)	3 MB	6 MB	73 MB
Once Tabling	3 MB	6 MB	73 MB

Table 3. Maximum table size for running Tilde on Mutagenesis.

Experiment	Lookahead	
	0	1
No Optimization	62094851	629636203
Prefix Tabling (Local)	17826104	122158703
Prefix Tabling (Batched)	17826104	122157269
Query Tabling (Local)	12878382	52005387
Query Tabling (Batched)	10436092	50486078
Once Tabling	33541719	287747657
Query Packs	17659174	30388904

Table 4. Total number of goal calls for running Tilde on Carcinogenesis.

Experiment	Lookahead	
	0	1
Prefix Tabling (Local)	1.5 MB	241 kB
Prefix Tabling (Batched)	1.5 MB	241 kB
Query Tabling (Local)	283 MB	365 MB
Query Tabling (Batched)	25 MB	257 MB
Once Tabling	20 MB	255 MB

Table 5. Maximum table size for running Tilde on Carcinogenesis.

Experiment	Lookahead		
	0	1	2
No Optimization	5007638	149930892	2625159941
Prefix Tabling (Local)	1699134	72049883	1180152456
Prefix Tabling (Batched)	1642281	72249388	1181515793
Query Tabling (Local)	6410317	-	-
Query Tabling (Batched)	1284685	70777264	-
Query Packs	1367278	36991076	418259748

Table 6. Total number of goal calls for running WARMR on Mutagenesis.

Experiment	Lookahead		
	0	1	2
Prefix Tabling (Local)	2 MB	2 MB	57 MB
Prefix Tabling (Batched)	2 MB	2 MB	57 MB
Query Tabling (Local)	776 MB	-	-
Query Tabling (Batched)	130 MB	882 MB	-

Table 7. Maximum table size for running WARMR on Mutagenesis.

lookahead setting. The once tabling approach always performs at least twice as good as the non-transformed queries. However, in most cases, it is still outperformed by the other optimizations. Because the tables are cleared in every iteration, the prefix tabling approach has very limited memory overhead, as can be seen in Tables 3 and 5. The total size of the tables used during once tabling does not differ much from the table sizes of query tabling. This confirms that execution seldom computes more than one solution for the tabled prefixes.

The results are even more pronounced when using WARMR, as can be seen in Table 6. Unfortunately, the machine on which the experiments were conducted (a Pentium 4 with 2 Gb RAM) ran out of memory for some experiments due to the size of the tables of the queries. This illustrates that the tabling approach can become a problem in practice. Table 6 does not include results of the once table experiments, because these could not be performed because of a bug in the YAP system at the time of running these experiments.

7 Conclusions

In this paper, we evaluated the application of tabled execution in the context of Inductive Logic Programming. Besides a description of the prefix and query tabling approaches presented in [7], we introduced once tabling as a tradeoff between storing all solutions and executing parts of queries multiple times. Instead of storing arbitrary many solutions of queries, the once tabling approach only

saves the first solution, at the price of potential recomputation if this solution does not suffice.

Although prefix tabling theoretically has similar effects than query pack execution, query packs always save at least as much calls as prefix tabling, without introducing the extra memory overhead. Query tabling, on the other hand, can potentially save more calls than query packs. Experiments pointed out that this is indeed the case for simple experiments, but that query packs exploit similarity of queries even further for larger experiments, and as such save more calls than query tabling. The once tabling approach requires only marginally less memory than the full query tabling approach. Although this approach yields less goal calls than unoptimized query execution, it performs worse than prefix tabling on most experiments. While the extra memory cost induced by the tabling approaches is still low enough for small data sets and simple queries, the size of the query tables grows to an unmanageable amount for more complex experiments.

From the conducted experiments, we conclude that tabling approaches do not apply well in the ILP setting. Not only does tabling fail to scale with larger problems because of the memory overhead, it is outperformed by more refined execution mechanisms such as query packs, which do not introduce an increasing memory overhead with increasing amounts of data.

Acknowledgements

We are indebted to the authors of [7] for providing us the implementation of their prefix and query tabling approaches.

References

1. ACE. The ACE data mining system, 2006. http://www.cs.kuleuven.be/~dtai/ACE/.
2. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
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 Research*, 16:135–166, 2002. http://www.cs.kuleuven.be/cgi-bin-dtai/publ_info.pl?id=36467.
4. L. Damas and V. S. Costa. YAP user's manual, 2003. http://yap.sourceforge.net.
5. L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
6. hipP. hipP: A high performance Prolog system, 2006. http://www.cs.kuleuven.be/~dtai/hipp/.
7. R. Rocha, N. A. Fonseca, and V. S. Costa. On Applying Tabling to ILP. In *Proceedings of the 16th European Conference on Machine Learning, ECML-05*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2005.
8. A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.

9. A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.

Embedding Solution Preferences via Transformation

Hai-Feng Guo and Miao Liu

Department of Computer Science
University of Nebraska, Omaha, NE 68182-0500
{haifengguo, miaoliu}@mail.unomaha.edu

Abstract. Preference logic programming (PLP) is an extension of constraint logic programming for declaratively specifying problems requiring optimization or comparison and selection among alternative solutions to a query. PLP essentially separates the programming of a problem itself from the criteria specification of its optimal solution. The main challenge to implement a PLP system is that how the defined solution preferences take effects automatically on pruning suboptimal solutions and their dependents during the computation. The solution preferences are specified at the Prolog programming level, whereas the answer collection is implemented at the system level. In this paper, we present a novel strategy to bridge the programming gap by introducing a new built-in predicate, which serves as the interface between the specification and the actual application of solution criteria. With this interface predicate, we can easily transform a preference logic program into an executable program, where solution preferences can be propagated into recursion so that selecting an optimal solution to a problem only depends on the optimal solutions to its subproblems. The strategy has been successfully implemented on a tabled Prolog system; and experimental results are also presented.

1 Introduction

Traditional constraint logic programming (CLP) specifies an optimization problem by using a set of constraints and minimizing (or maximizing) an objective function. Unfortunately, general optimization problems may involve compound objectives whose optima are difficult to be represented by a simple minimization (or maximization). Even worse, for many applications, especially those defined over structural domains, it is difficult to specify any objective function. For this reason, an extension of constraint logic programming called preference logic programming (PLP [8,9]) has been introduced for declaratively specifying problems requiring optimization or comparison and selection among alternative solutions to a query. The PLP paradigm essentially separates the constraints of a problem itself from its optimization or selection criteria, and makes optimization or selection a meta-level operation. Preference logic programming has been shown useful for many practical applications such as in artificial intelligence [2], data mining [4], document processing and databases.

We proposed a method in [13] for specifying preference logic programs using tabled Prolog programs. The method follows the PLP paradigm to separate the constraints of a problem itself from its optimization or selection criteria. Each preference logic program is defined using two disjoint sets of definite clauses, where one contains the specification of the constraints of the problem and the other defines the optimization or selection criteria. Connection between these two sets is established by a mode declaration scheme [11]. For instance, a preference logic program for finding the shortest path may be defined as Example 1, where the optimization predicate `path/5` contains a general definition, clauses (1-3), and two solution preferences, clauses (5-6). The informal meaning of clause (5) is that `path(X,Y,C2,D2,_)` is preferred than `path(X,Y,C1,D1,_)` if `C2` is less than `C1`, which tells that a lower-cost path is preferred. Clause (6) tells that if two paths have the same costs, then a shorter path is preferred. Clause (4) serves as the connector between the general definition and the solution preferences, designating `path/5` as an optimization predicate subject to the selection criteria defined in the '`<<<`'-clauses, where '`<<<`' is an infix binary predicate used specifically for defining the preferences.

Example 1. Consider the following preference logic program searching for a lowest-cost path; if two paths have the same cost, then the one with a shorter distance is preferred. Predicate `path(X,Y,C,D,L)` denotes a path from `X` to `Y` with the cost `C`, the distance `D` and the path route `L`.

```
path(X, X, 0, 0, []).                                   (1)
path(X, Y, C, D, [(X, Y)]) :- edge(X, Y, C, D).         (2)
path(X, Y, C, D, [(X, Z) | P]) :-
    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),
    C is C1 + C2, D is D1 + D2.                         (3)

:- table path(+, +, <<<, <<<, -).                       (4)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :- C2 < C1.     (5)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C1 = C2, D2 < D1.                                   (6)
```

We use a tabled Prolog system [12] to implement preference logic programs, because a tabled Prolog can be thought of as an engine for efficiently computing fixed points, which is critical for finding the model-theoretic semantics of a preference logic program. Additionally, a new mode-declaration scheme for tabled Prolog systems has been proposed in [11] that provides an attractive platform for specifying dynamic programming problems in a simpler manner: 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 Prolog system that uses the appropriate mode declaration. This mode-declaration scheme can be further extended for specifying and executing preference logic programs, since the mode-declaration scheme is indeed the intent of PLP, that is, tabled Prolog

systems can selectively choose "better" answers for a given tabled predicate call guided by the declared mode, while PLP selectively chooses "better" solutions based on preferences.

The main challenge to implement a PLP system is how the defined solution preferences take effects automatically on pruning suboptimal solutions and their dependents during computation. The solution preferences are specified at the Prolog programming level, whereas the answer collection is implemented at the system level.

In our previous work [13], the computation of a preference Prolog program is achieved in two steps. First, a mode-directed automatic transformation is applied to embed the preferences into the problem specification. Second, the transformed program is then evaluated using tabled resolution, while the mode declaration provides a selection mechanism among the alternative solutions. However, this computation strategy has a strict assumption that the defined preference relation '<<<' is a total order relation. That is, every two atoms of an optimization predicate are comparable, and the problem has only a single optimal solution.

In this paper, we present a novel strategy to bridge the programming gap by introducing a new built-in predicate, which serves as the interface between the specification and the actual application of solution criteria. With this interface predicate, we can easily transform a preference logic program into an executable program, where solution preferences can be propagated into recursion so that selecting an optimal solution to a problem only depends on the optimal solutions to its subproblems. Furthermore, the new strategy gets rid of the assumption of a total-order preference relation. Instead, it can support preference logic programs with any relational preferences and multiple optimal solutions.

The rest of the paper is organized as follows: Section 2 gives a brief introduction on a tabled Prolog system and the mode declaration scheme for tabled predicates; section 3 introduces the syntax of a preference logic program; section 4 presents our previous strategy to transforming a preference logic program to an executable tabled Prolog program; section 5 shows our new simplified transformation strategy with a new builtin predicate; section 6 addresses some efficiency issues related to implementation; and section 7 gives our conclusions.

1.1 Other related work

The concept of preference often occurs in the soft constraint programming paradigm. Soft constraints [1,7,15,16,6,17] were proposed to mainly handle the over-constrained problems, where no solution exists due to conflict among constraints, or a satisfiable problem where many solutions are equally proposed by the system. Using soft constraint techniques, such as fuzzy constraint satisfaction, partial constraint satisfaction, or hierarchical constraint satisfaction, solutions are always found and ranked according to the optimization criteria. Such soft constraints usually take preferences over constraints. Hierarchical constraint logic programming [17] proposed preferences on constraints indicating the preference strength of each constraint, which gives great flexibility for specifying those constraint problems with non-strict requirements. Rossi and Sperduti [14] made it possible

to support both constraint and solution preferences, where preferences are specified via setting preference ranks (low, medium, high and needed) over variable values and constraints, and then soft solution constraints are interactively generated through a machine learning procedure. Another approach for describing preferred criteria in CLP is given in the problems of relational optimization [5], where a preference relation indicates when a solution is better than another solution. However, this preference relation is limited to a total order relation, i.e., every two solutions in a problem domain have to be comparable.

In this paper, we focused on Prolog programming with solution preferences, where the defined preference relation has no special order restriction.

2 Tabled Prolog with Mode Declaration

2.1 Tabled Prolog

Traditional Prolog systems use SLD resolution [10] 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 Prolog [3,19,12,18] eliminates such infinite loops by extending logic programming with tabled resolution. The main idea is to memoize the answers to some calls and use the memoized 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. Thus, a tabled Prolog system can be thought of as an engine for efficiently computing fixed points.

In a tabled Prolog 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 re-computation. Consider Example 2 checking the reachability relation. This program does not work properly in a traditional Prolog system. With the declaration of a tabled predicate `reach/2` in a tabled Prolog system, it can successfully find a set of complete solutions `X=b`, `X=c` and `X=a`, due to the fixed-point computation strategy, albeit the predicate is defined left-recursively.

Example 2. Consider the following tabled Prolog program checking the 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).
```

2.2 Mode Declarations

However, the fixed point of a computing model may contain infinite number of solutions, which certainly affects the completion of the computation. Consider Example 3 searching the paths for reachable nodes. An extra argument is added for the predicate `reach/3` to record the found path. However, this extra argument results in the fixed point of the computation to be infinite, since there is an infinite number of paths from `a` to any node due to the cycle between `a` and `b`. Therefore, a meta-level operation is useful to filter the infinite-size solution set to a finite one so that the computation can be completed.

Example 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,a)]).
:- reach(a,X,E).
```

This meta-level operation can be achieved by a mode declaration for a tabled predicate, which is described in the form of

:- table $q(m_1, ..., m_n)$.

where q/n is a tabled predicate name, $n \geq 0$, and each m_i has one of the forms as defined in Table 1.

Modes	Informal Semantics
$+$	an indexed argument
$-$	a non-indexed argument
last	a non-indexed argument for the last answer
`<<<`	a user-defined preference mode

Table 1. Built-in Modes for Tabled Predicates

The mode declaration [11] was initially used to classify arguments as indexed (`+`) or non-indexed (`-`) for each tabled predicate. Only indexed arguments are used for variant checking during collecting new generated answers into the table. For a tabled call, any answer generated later for the same value of the indexed arguments is discarded because it is a variant (w.r.t. the indexed arguments) of

a previously tabled answer. This step is crucial in ensuring that a fixed-point is reached. Consider again the program in Example 3. Suppose we declare the mode as "`:- table reach(+,+,-)`"; this means that only the first two arguments of the predicate `reach/3` are used for variant checking. As a result, the computation can be completed properly with three answers, that is, each reachable node from `a` has a simple path as an explanation.

The mode directive `table` makes it very easy and efficient to extract explanation for tabled predicates. In fact, our strategy of ignoring the explanation argument (mode '`-`') 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 also ensures that those generated explanations are concise and that cyclic explanations are guaranteed to be absent. More importantly, the meta-level mode declaration is especially useful to reduce an infinite computation model to a finite one for some practical uses, or a big computation model to an optimized smaller one.

The mode directive `table` can be further extended to associate a non-indexed argument of a tabled predicate with some optimum constraint. With the mode '`-`', a non-indexed argument for each tabled answer only records the very first instance. This "very first" property can actually be generalized to support any other preferences, e.g., the minimum/maximum value with mode `min`/`max`, etc. Contrary to the mode '`-`', the mode '`last`' is useful for recording the last answer from a solution set. This mode can be realized as follows: for a tabled call, whenever a new answer is generated, it will replace the old tabled one, if any, so that the tabled answer is always the 'last' answer generated so far. The mode `<<<` is used to support user-defined preferences as described later.

3 Programming with Solution Preferences

In optimization problems, we are often interested in comparing alternative solutions to a set of constraints and choosing the "best" one. In general, there are two components of an optimization problem: (i) specification of the constraints to the problem; and, (ii) specification of which predicates to be optimized and how the optimal solutions are selected. The intent of preference logic programing is to separate these two components and declaratively specify such applications [13]. We give the definition of a preference logic program in the paradigm of tabled Prolog programming.

Definition 1 (Preference Logic Programs). *A (definite) preference logic program P can be defined as a pair $<P_{core}, P_{pref}>$, where P_{core} and P_{pref} are two syntactically disjoint sets of clauses defined as follows: P_{core} specifies the constraints of the problem using a set of definite clauses; P_{pref} defines the optimization criteria using a set of preference clauses (or preferences) in the form of:*

$$p(\tilde{T}_1) \texttt{ <<< } p(\tilde{T}_2) \texttt{ :- } A_1, A_2, \cdots, A_n. \qquad (n \geq 0)$$

where p can be any user-defined predicate in P_{core}, $p(\tilde{T}_1)$ and $p(\tilde{T}_2)$ have the

same arity, and we call such a predicate p an optimization predicate*; each* A_i $(1 \leq i \leq n)$*, independent from any defined optimization predicate, is an atom defined in P; and* `<<<` *is simply an infix binary predicate especially used in a preference clause.*

The informal semantics of $p(\tilde{T_1})$ `<<<` $p(\tilde{T_2})$ `:-` $A_1, A_2, \cdots, A_n$ is that the atom $p(\tilde{T_1})$ is less preferred than $p(\tilde{T_2})$ if $A_1, A_2, \cdots, A_n$ are all true. Note that the two atoms being compared have the same predicate name and arity; and we use $\tilde{T}$ to represent a tuple of terms. We say that A_i is independent from any defined optimization predicate, which means the truth value of A_i does not depend on any subgoal of defined optimization predicates. The reason of this restriction is consistent the essential idea of PLP, separating the criteria for selecting better answers from their definition.

Consider the Example 1 again. This optimization problem with compound objectives is relatively difficult to solve using traditional constraint programming. A two-step selection procedure is usually involved, where firstly only the cost criterion is used to find all the lowest-cost paths, and secondly the optimal path is selected by comparing distances among the lowest-cost paths. However, the preference logic program, as shown in Example 1, is intended to specify and solve this problem directly in a declarative method. It separates the constraints of a problem itself from the criteria for selecting the optimal solutions, so that compound optimization criteria can be easily added in an incremental way. Clauses (1) to (3) make up the core program P_{core} defining the path relation and a directed graph with a set of edges; clauses (4) to (6), the preference part P_{pref}, specify the predicate `path/5` to be optimized and give the criteria for optimizing the `path/5` predicate. The responsibility of how to find the optimal solution is shifted to the underlying logic programming system, in keeping with the spirit of logic programming as a declarative paradigm.

4 Embedding Preferences via A Naive Transformation

We start with a transformed example, and then present the formal transformation rules.

Example 4. Consider the following tabled program transformed from the program in Example 1.

```
pathNew(X, X, 0, 0, []).                          (1)
pathNew(X, Y, C, D, [e(X, Y)]) :-
    edge(X, Y, C, D).                             (2)
pathNew(X, Y, C, D, [e(X, Z) | P]) :-
    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),
    C is C1 + C2, D is D1 + D2.                   (3)

:- table path(+, +, last, -, -).                  (4)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
```

```
    C2 < C1.                                         (5)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C1 = C2, D2 < D1.                                (6)

path(X, Y, C, D, P) :-
   pathNew(X, Y, C, D, P),
   ( path(X, Y, C1, D1, P1)
      -> path(X,Y,C1,D1,P1) <<< path(X,Y,C,D,P)
      ;  true ).                                     (7)
```

Three major changes have been made in this transformation by taking advantage of the unique global *table* in the system: (i) The original predicate `path/5` in Example 1 is replaced by a new predicate `pathNew/5` to emphasize that this predicate generates a new preferred path candidate from `X` to `Y`. (ii) The predicate `path/5`, given a new definition in clause (7), represents the way for identifying a preferred answer. The meaning of clause (7) is the following: given a path candidate A by `pathNew(X,Y,C,D,P)`, we need to check whether there already exists a tabled answer, if so, they are compared with each other to keep the preferred one in the table; otherwise, the candidate is recorded as a first tabled answer. By this way, the optimal solution must be the *last* one left in the table after all the computation and comparisons are done. Notice that in a tabled Prolog system, the answer will be automatically recorded into the table once the subgoal `path(X,Y,C,D,P)` succeeds. (iii) The first preference mode '`<<<`' for `path/5` is hence replaced by '`last`' to catch the last and optimal tabled answer, while the rest of preference modes are replaced by '`-`', since all those arguments are bundled together in their preference rules. The formal transformation rules are given in Definition 2.

Definition 2 (ρ-transformation). *Let P be a preference program. The transformation to a new tabled program $P' = \rho(P)$ can be formalized as follows:*

– *for any optimization predicate q/n in P, we have*
  - *for each clause defining q/n in P, the predicate q/n in its head (the part to the left of :-) is renamed to a new predicate q'/n;*
  - *a new clause definition for q/n is introduced in the form of:*

    $q(a_1, \cdots, a_n)$ `:-`
    $\quad q'(a_1, \cdots, a_n)$,
    $\quad ($ $q(b_1, \cdots, b_n)$
    $\qquad$ `->` $q(b_1, \cdots, b_n)$ `<<<` $q(a_1, \cdots, a_n)$
    $\qquad$; `true`).

    where $a_1, \cdots, a_n$ and $b_1, \cdots, b_n$ are all variables; for $1 \leq i, j \leq n$, a_i and b_i use an identical variable if ith argument of q/n is an indexed one with the mode '`+`'; a_i and a_j (or b_i and b_j) use two different variables if $i \neq j$;
  - *in the mode declaration of q/n, the first preference mode '`<<<`' is changed into '`last`', and the rest preference modes (if any) are changed into the standard non-indexed mode '`-`';*

– *for the other clauses not defining optimization predicates in P, make a same copy into P'.*

We call this transformation ρ-transformation.

The main advantage of the ρ-transformation is that the transformed program is executable in a tabled Prolog environment. However, it has the following limitations on preference logic programming. It is required that preferences are well defined such that the preference relation `<<<` is a total order relation. That is, (1) every two atoms (e.g., α and β) of an optimization predicate are comparable (α `<<<` β or β `<<<` α) in terms of the preference relation. (2) Contradictory preferences (e.g., $p(a)$ `<<<` $p(b)$ and $p(b)$ `<<<` $p(a)$) are not allowed in the programs. (3) It is assumed that there is only a single optimal answer. All those limitations prevent preference logic programming from broad practical applications.

5 Embedding Preferences via An Interface Predicate

The main challenge for preference logic programming is how we can use preferences to automatically prune suboptimal solutions and its dependents. A preference logic program cannot be easily supported on any existing tabled Prolog system. The main reason is that the mode-controlled table manipulation is implemented at the system level, whereas preferences are defined at the Prolog programming level. Therefore, we introduce the idea of an interface predicate to connect the system level table manipulation and the Prolog-level preferences so that the preferences can be used as the filter criteria automatically for the answer collection at the system implementation level.

5.1 The Interface Predicate

A new builtin Prolog predicate, named `updateTable(Ans)`, is introduced to serve the purpose of an interface between the table manipulation and preferences, where `Ans` is a new potential answer to a subgoal of an optimization predicate. Whenever a new answer `Ans` is generated to a subgoal of optimization predicate, an instance subgoal of `updateTable/1` is invoked. It determines whether the obtained new answer `Ans` is a preferred atom or not. The subgoal of `updateTable/1` succeeds if `Ans` is a preferred atom in the table. Otherwise, the `updateTable/2` subgoal will fail and `Ans` will be discarded. Another important purpose of `updateTable/1` is that it removes any existing tabled answers which are less preferred than `Ans`. Therefore, the interface predicate `updateTable/1` maintains the property at runtime that every tabled answer is a preferred atom.

As shown in Figure 1, the `updateTable/1` procedure starts with a subgoal `allTabledAnswers(TabAnsList)`, which retrieves all of the tabled answers of the corresponding subgoal into the variable `TabAnsList`, and then calls a subgoal `updateTabledAnswers(TabAnsList, Ans)`, which updates the tabled answers properly based on comparing the current answer `Ans` with the existing ones

```
updateTable(Ans) :-                                          (1)
    allTabledAnswers(TabAns),                                (2)
    updateTabledAnswers(TabAns, Ans).                        (3)

updateTabledAnswers(TabAns, Ans) :-                          (4)
    compareToTabled(TabAns, Ans, Flist, AnsFlag),            (5)
    removeTabledAnswers(Flist),                              (6)
    AnsFlag == 0.                                            (7)

compareToTabled([], _, [], Flag) :-                          (8)
    (var(Flag) -> Flag = 0 ; true ).                         (9)
compareToTabled([T|Ttail], Ans, [F|Ftail], Flag) :-         (10)
    (T <<< Ans -> F = 1 ; F = 0),                           (11)
    (Ans <<< T -> Flag = 1 ; true),                         (12)
    compareToTabled(Ttail, Ans, Ftail, Flag).               (13)
```

Fig. 1. Definition of `updateTable/1`

in `TabAnsList`. The subgoal `compareToTabled/4`, as shown in lines 8 to 13, compares the new generated `Ans` against each of the tabled solution based on the optimization criteria `<<<`. The variable `Flist`, consisting of a list of boolean values, represents the comparison results and tells whether each tabled answer is a preferred atom among the current tabled answers. The boolean value `1` means that the corresponding tabled answer is a suboptimal one, and therefore the answer will be removed in the subgoal `removeTabledSolutions(Flist)`. Similarly, the boolean variable `Flag` tells whether the current answer `Ans` is a preferred atom among the current tabled answers. Line 7 guarantees that the subgoal of `updateTable/1` succeeds only if the obtained answer `Ans` is a preferred atom at this computation point. The answer `Ans` will be automatically inserted into the table after `updateTable(Ans)` succeeds.

The predicates `allTabledAnswers/1` and `removeTabledAnswers/1` are recently implemented at the system level (C programming) for manipulating tabled answers.

5.2 Transformation

Example 5 shows the new transformed program with a new interface predicate `updateTable/1`. The transformation is significantly simplified compared to the the previous one in Example 4. The main change in the new transformation is the addition of the subgoals (`updateTable/1`) to each definition of optimization predicate `path/5`. These `updateTable/1` subgoals are invoked every time a defined clause of `path/5` is completely executed and a potential answer is obtained. Therefore, it is the right point for `updateTable/1` to compare the new potential answer with the tabled answers based on the solution preferences, and then update the table properly.

Example 5. Consider the following tabled program transformed from the program in Example 1 using the interface predicate `updateTable/1`.

```
path(X, X, 0, 0, []) :-
    updateTable(path(X,X,0,0,[])).                    (1)
path(X, Y, C, D, [e(X, Y)]) :-
    edge(X, Y, C, D),
    updateTable(path(X,Y,C,D,[e(X,Y)])).              (2)
path(X, Y, C, D, [e(X, Z) | P]) :-
    edge(X, Z, C1, D1), path(Z, Y, C2, D2, P),
    C is C1 + C2, D is D1 + D2,
    updateTable(path(X,Y,C,D,[e(X,Z)])).              (3)

:- table path(+, +, <<<, <<<, -).                     (4)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C2 < C1.                                          (5)
path(X,Y,C1,D1,_) <<< path(X,Y,C2,D2,_) :-
    C1 = C2, D2 < D1.                                 (6)
```

The new transformation is formally given in Definition 3. The main task is simply to append a subgoal `updateTable/1` to the end of each clausal definition of the optimization predicate. Furthermore, the transformed preference program does not require the preference relation '`<<<`' to satisfy the total order relation.

Definition 3 (η-transformation). *Let P be a preference program. The transformation to a new tabled program $P' = \eta(P)$ can be formalized as follows:*

– *for each clausal definition "H `:-` $B_1, \cdots, B_n$." (where $n \geq 0$) of an optimization predicate in P, we have a new definition in p' as follows:*
$$H \texttt{ :- } B_1, \cdots, B_n,\ updateTable(H).$$
– *for the other clauses not defining optimization predicates in P, make a same copy into P'.*

*We call this transformation η-*transformation.

5.3 Flexibility

Preference logic programming with η-transformation can specify optimization problems with more generalized preferences. A total order relation is not a necessary condition for the preference relation `<<<`. If two atoms of an optimization predicate are not comparable in terms of the preference relation, then both of them can be preferred atoms kept in the table. Thus, multiple optimal answers are supported during the computation. On the other hand, if contradictory preferences (e.g., $p(a)$ `<<<` $p(b)$ and $p(b)$ `<<<` $p(a)$) occur, then both of them should be removed from the table, which is consistent to its intended preference model.

The mode-directed preferences provides flexible and declarative meta-level controls for logic programmer to extend the general specification of a problem to an optimization specification. Such an extension can often reduce a general problem with a big solution set to a refined problem with a small preferred solution set. Consider again the lowest-cost path problem in Example 1. Clauses (1) to (3) define a general path with cost, distance and evidence information; clauses (5) and (6) gives user-defined preferences. The mode declaration

```
:- table path(+, +, <<<, <<<, -)
```

basically maps the general path definition to a preferred path definition, where if multiple preferred paths with the same cost and distance exist, then only the first computed one will be recorded in the table due to the mode '`-`'. However, if all possible preferred paths are needed, the user can specify the mode declaration as follows:

```
:- table path(+, +, <<<, <<<, <<<),
```

which requires the support of collecting multiple optimal answers from a preference logic programming system.

In addition, Example 1 actually shows a dynamic programming approach for the shortest path problem. Once the predicate `path/5` is declared as an optimization predicate with preferences, the answers to any predicate call of `path/5` represent optimal solutions. More importantly, those preferences are not only applied on the top query of the optimization predicate, but also propagated into its recursive subgoals. As a result, an optimal solution is defined recursively based on the optimal solutions to its subproblems. This scheme is different from the one in the naive shortest path algorithm which enumerates all possible paths from `X` and `Y` and then compares them to find the optimal one.

6 Experimental Results

Our experimental benchmarks for preference logic programming include five typical optimization 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 CPU 2.4GHz machine with 512M RAM running RedHat Linux 9.0.

The experimental results show that preferences provide a declarative approach without sacrificing efficiency of dynamic programming. Table 2 compares the running time performance between the programs with and without preferences. For the preference programs, the tabled system collects optimal answers implicitly by applying the predefined preferences; the programs without preferences adopt a traditional method – e.g., use the builtin predicate `findall/3` – to collect all the possible answers explicitly and then locate the optimal one at every recursive stage. The experimental data indicates, based on the running timings and their ratios in Table 2, that the programs with preference declaration are better than or comparable to those corresponding programs without preference declaration.

	matrix	lcs	obst	apsp	knap
without preferences	1.47 (1.0)	0.80 (1.0)	1.50 (1.0)	3.14 (1.0)	99.69 (1.0)
Preferences with A Naive Transformation	0.37 (0.25)	0.53 (0.66)	0.38 (0.25)	3.11 (0.99)	77.45 (0.78)
Preferences with An Interface Predicate	1.06 (0.72)	0.70 (0.88)	1.11 (0.74)	2.63 (0.84)	58.96 (0.59)

Table 2. Running time performance comparison: Seconds/(Ratio)

The efficiency for preference programming are mainly credited to the following two points. First, tabled Prolog systems with mode declaration provides a concise yet easy-to-use interface for preference logic programming. The transformation procedure to incorporate the problem specification and preferences does not introduce any major overhead; mode declarations are flexible and powerful for supporting user-defined preferences, and the mode functionality is implemented at the system level instead of the Prolog programming level. Second, those preferences are not only applied on the top query of the optimization predicate, but also propagated into its recursive subgoals automatically.

A disadvantageous efficiency issue is the frequent retrieval or replacement of tabled answers. That is because the optimized answer is dynamically selected by comparing with old tabled answers according to the preferences. The retrieval of a tabled answer for comparison incurs time overhead due to having to locate each argument of the answer in the table. For replacing a tabled answer in the current TALS system, if a 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.

For preference logic programs, the running timings in the new strategy (with an interface predicate) and the old strategy (with a naive transformation) are comparable. However, the reason to their efficiency differences is not clear to us. The important point is that preferences with the interface predicate can support more generalized optimization problems than the previous one with the naive transformation.

7 Conclusion

In this paper, we presented a new declarative strategy to embed solution preferences into a program definition, so that the given preferences can take effects automatically on pruning suboptimal solutions and their dependents during the computation. The strategy has been implemented successfully in a tabled Prolog

system for supporting solutions preferences. The main difficulty of supporting preferences lies on how to bridge the gap between their specification at the Prolog programming level and their actual functionality at the system implementation level. Our new strategy introduces a new built-in predicate, which changes the traditional way of collecting tabled answers blindly into a selective way by applying preference criteria. With this interface predicate, we can easily transform a preference logic program into an executable program, where solution preferences can be propagated into recursion so that selecting an optimal solution to a problem only depends on the optimal solutions to its subproblems. In addition, the new strategy relaxes the total order relation on the preference predicate to allow that the preference relation may be contradictorily defined, or that not every two solutions are comparable. Therefore, preference logic programming with the new transformation strategy provides more flexibility in specifying and solving more optimization problems.

References

1. S. Bistarelli, U. Montanari, and F. Rossi: Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.
2. A. Brown, S. Mantha, and T. Wakayama: Preference Logics: Towards a Unified Approach to Non-Monotonicity in Deductive Reasoning. *Annals of Mathematics and Artificial Intelligence*, 10:233–280, 1994.
3. Weidong Chen and David S. Warren: Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1), pp. 20–74, 1996.
4. Baoqiu Cui and Terrance Swift: Preference Logic Grammars: Fixed Point Semantics and Application to Data Standardization. *Artificial Intelligence*, 138(1-2): 117–147, 2002.
5. F. Fages: On the Semantics of Optimization Predicates in CLP Languages. In *Proc. 13th FST-TCS*, 1993.
6. H. Fargier and J. Lang: Uncertainty in constraint satisfaction problems: a probabilistic approach. European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty, pp. 97–104, 1993.
7. Eugene C. Freuder and Richard J. Wallace: Partial Constraint Satisfaction. 11th International Joint Conference on Artificial Intelligence, pages 278–283, 1989.
8. K. Govindarajan, B. Jayaraman, and S. Mantha: *Preference Logic Programming.* International Conference on Logic Programming (ICLP), pages 731–745, 1995.
9. K. Govindarajan, B. Jayaraman, and S. Mantha: Optimization and Relaxation in Constraint Logic Languages. *POPL* 1996: 91–103.
10. J.W. Lloyd. Foundations of Logic Programming. Springer-Verlag, 1987.
11. Hai-Feng Guo and Gopal Gupta: Simplifying Dynamic Programming via Tabling. *Practical Aspects of Declarative Languages (PADL)*, pages 163–177, 2004.
12. 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.
13. H-F. Guo, B. Jayaraman, G. Gupta, and M. Liu: *Optimization with mode-directed preferences.* International Conference on Principles and Practice of Declarative Programming, pages 242-251, 2005.

14. F. Rossi and A. Sperduti: Acquiring both constraint and solution preferences in interactive constraint systems. *Constraints*, 9(4):311–332, 2004.
15. Zs. Ruttkay: Fuzzy Constraint Satisfaction. *Proc. 3rd IEEE International Conference on Fuzzy Systems*, 1994.
16. T. Schiex: Arc consistency for soft constraints. International Conference on Principles and Practice of Constraint Programming, 2000.
17. M. Wilson and A. Borning: *Hierarchical Constraint Logic Programming*. Journal of Logic Programming, 16:277–318, 1993.
18. R. Rocha, F. Silva, and V. S. Costa: On a Tabling Engine That Can Exploit Or-Parallelism. In *ICLP* Proceedings, pages 43–58, 2001.
19. Neng-Fa Zhou, Y. Shen, L. Yuan, and J. You: Implementation of a Linear Tabling Mechanism. *Journal of Functional and Logic Programming*, 2001(10), 2001.

Towards Region-based Memory Management for Mercury Programs ⋆

Quan Phan and Gerda Janssens

Department of Computer Science, K.U.Leuven
Celestijnenlaan, 200A, B-3001 Heverlee, Belgium,
{quan.phan,gerda.janssens}@cs.kuleuven.be

Abstract. Region-based memory management is a form of compile-time memory management, well-known from the functional programming world. This paper describes region-based memory management for the Mercury language using some points-to graphs that model the partition of the memory used by a program into separate regions and distribute the values of the program's variables over the regions. First, a region analysis determines the different regions in the program. Second, the liveness of the regions is computed. Finally, a program transformation adds region annotations to the program for region support. Our approach obtains good results for a restricted set of deterministic Mercury programs and is a valid starting point to make region-based memory management work with general Mercury programs.

1 Introduction and Motivation

Logic programming (LP) languages aim to free programmers from procedural details such as memory management tasks. One classical, automatic memory management technique in logic programming is to use a heap memory for structured terms and rely on backtracking and runtime garbage collection to reclaim memory. While efficient implementations of garbage collectors for LP languages can reuse more than 90% of the heap space of a program, they incur execution overhead because collectors often need to temporarily stop the main program.

To remedy this shortcoming compile-time techniques automatically enhance programs with instructions to reuse memory. This static method generally follows two approaches: region-based memory management (RBMM) and compile-time garbage collection (CTGC). The basic idea of RBMM is to divide the heap memory used by a program into different regions. The dynamically created terms and their subterms have to be distributed over the regions in such a way that at a certain point in the execution of the program all terms in a region are dead and the region can be removed. RBMM is a topic of intensive research for functional programming languages [13,1,4,12] and more recently also for imperative languages [3,2]. For LP languages, there has been only one attempt to make RBMM work for Prolog [7,6,5]. CTGC detects for a program when allocated

⋆ This work is supported by the project GOA/2003/08 and by FWO Vlaanderen.

memory cells are no longer used and instructs the program to reuse those cells for constructing new terms, reducing the memory footprint and in some cases achieving faster code. This idea has been used to reuse memory cells locally in the procedures of Mercury programs [9,10,8].

Our ultimate research goal is to investigate the possibility and practicality of a hybrid memory management technique, which combines RBMM and CTGC to capitalize on their advantages. As CTGC was developed for Mercury, relying much on Mercury-specific information such as determinism, modes, and types, which are not readily available in other LP systems it is a logical choice to develop RBMM in the same context. Moreover, this information can also be exploited in our model of the heap when it is divided into regions and in the region analyses.

The contribution of this paper is to develop an automated system based on program analysis that adds region annotations to Mercury programs. While program analyses based on points-to graphs are popular for languages with destructive update, in LP our work is the first to use the points-to graph to model the heap and to develop the program analyses that derive the model and make use of it for memory management.

In Section 2 we explain how memory management for Mercury can be based on the use of regions. The whole algorithm is composed of three phases, which are described in Sections 3, 4, and 5, respectively. Section 3 introduces the concept of the region points-to graph and the points-to analysis. Section 4 defines the live region analysis which uses the region points-to graph of each procedure to precisely detect the lifetime of the regions. We do the transformation by adding RBMM annotations in Section 5. Section 6 discusses the results of our prototype analysis system. Finally, Section 7 concludes.

2 Regions and Mercury

2.1 Mercury programs

We assume that the input of our program analysis is a Mercury program that is transformed by the MMC into *superhomogeneous* form, with the goals reordered and each unification specialized into a *construction* (<=), a *deconstruction* (=>), an *assignment* (:=), or a *test* (==) based on the modes [11]. The *qsort* program then consists of the following procedures (Fig. 1)[1]:

2.2 Usage of Regions: An Example

We illustrate the usefulness of distributing terms over regions using the *qsort* example. In the implementation of LP languages, a list consists of the list skeleton and the elements. Observing the memory behaviour of the *qsort* procedure, we

[1] The bold text can be ignored as it is the region annotations added automatically by the transformation. The italic comments are the corresponding transformation rules, which are described later in Sect. 5.

```
main(!IO) :-
  create R1,
  create R2,
  (1) R1.L<=[R2.2,R2.1,R2.3],
  create R3,
  (2) R3.A<=[],
  (3) qsort(L,A,S){R1,R2,R3},
  (4) L is no longer used ...
  (5) ...

qsort(L,A,S){R1,R2,R3} :-
(
(1) R1.L=>[],
    remove R1, % due to T4
(2) S:=A
;
(3) R1.L=>[Le|Ls],
(4) split(Le,Ls,L1,L2){R2,R1,R2,R4,R5},
(5) qsort(L2,A,S2){R5,R2,R3},
(6) R3.A1<=[Le|S2],
(7) qsort(L1,A1,S){R4,R2,R3}
).

split(X,L,L1,L2){R5,R1,R2,R3,R4} :-
(
(1) R1.L=>[],
    remove R1, % due to T4
    create R3, % due to T2
(2) R3.L1<=[],
    create R4, % due to T2
(3) R4.L2<=[]
;
(4) R1.L=>[Le|Ls],
    (
    (5) R5.X>=R2.Le
    − >
    (6) split(X,Ls,L11,L2){R5,R1,R2,R3,R4},
    (7) R3.L1<=[Le|L11]
    ;
    (8) split(X,Ls,L1,L21){R5,R1,R2,R3,R4},
    (9) R4.L2<=[Le|L21]
    )
).
```

Fig. 1. *qsort* program.

see that the output list has a new skeleton built up in the accumulator while its elements are those of the input list. In the *main* predicate, the input list L is no longer used after the call to *qsort*. This means that if the skeleton of the input list, the elements, and the skeleton of the output list (and the accumulator) are stored in three different regions we can safely free the memory occupied by the input list's skeleton by removing its region after the call. Take a closer look inside the *qsort* procedure at (4). The call to *split* creates two new lists with two new skeletons while the elements are also those of the input list. Therefore, if the two new skeletons are stored in regions different from the region of the input list's skeleton, the region can even be removed after this call inside *qsort*. This means that we detect even an earlier point where the region can be removed. So, by storing different components of the lists in separate regions we can do timely removal, recovering dead memory sooner.

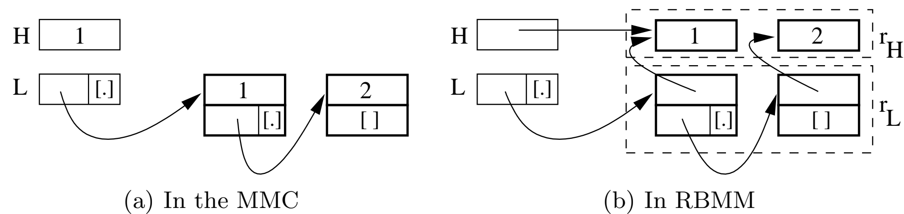
The *qsort* program with region support produced by our analysis is shown in Fig. 1 with $\{\ldots, R_i, \ldots\}$ the list of region parameters of a procedure. Two instructions *create* and *remove* manipulate the creation and removal of regions. By $R.X$ we denote the variable X and its region. Detailed information about the regions for the variables will be given in Fig. 3. In *qsort* procedure, the region of the skeleton of the list L passed to *qsort* from *main* is removed in the base case branch of *split* in the call at (4). The two new skeletons of the lists *L1* and *L2* are allocated in two separate regions, which are created in the base case branch of *split*. Those regions are removed in the calls at (5) and (7). If *L1* and *L2* are empty lists their regions are removed in the base case branch of *qsort*. Otherwise

they are removed in the base case branch of *split* the same as what happens to the input list L. The region of the output list's skeleton is the region of the accumulator, which is created in *main*.

2.3 Storing Terms in Regions

The way terms are allocated in memory depends on the specific implementation of a language. Therefore, we discuss the term representation when the heap memory is divided into regions in the Melbourne Mercury Compiler (MMC).

In the MMC, the way a term is stored depends on its type. Terms can be either of primitive types such as integer, char, float and string, or of discriminated union types. The latter types whose type constructors are functors with arity zero are called enumeration types, otherwise they are called record types. Primitive and enumeration terms are stored in a single memory cell. A term of a record type is stored on the heap and pointed to by a tagged pointer where the tag encodes the principal functor of the term and the pointer itself points to the block of memory cells corresponding to the arguments (subterms) of the principal functor. Fig. 2(a) shows the representation of a primitive type variable H bound to an integer and of a record type variable L bound to a list of two integers. A box with a slim border is a location on the stack or registers, one with a bold border is a location on the heap.



(a) In the MMC (b) In RBMM

Fig. 2. H=1, L=[H,2].

Now we consider the term representation when the heap is split into regions. To simplify the presentation in this paper, we assume that all terms are constructed on the heap and that the subterms of a compound term are always stored as if they are of record type. A primitive or enumeration term is stored in a memory cell in a region. To store a compound term, a region is used to store the memory block of its subterms. For a variable that is bound to the term, this region is called the region of the variable. If a subterm is of the same type as its compound term, it is stored in the same region as the compound term. Otherwise it is stored in a different region. Fig. 2(b) shows the division of the heap into regions by using dashed lines. The variable H is bound to an integer that is stored in the region r_H. The two-cell memory blocks making up the skeleton of the list to which L is bound are put into the other region, called the region of L

(r_L), because the second subterm of a list is also of type list. Also, the elements reachable from the skeleton in r_L are put in the same region, here r_H.

There are several good reasons for our choice. First, the representation ensures that the terms of a recursive type are always stored in a finite number of physical regions, regardless of the terms' actual sizes. Second, a program often treats the values of the same type in a term in the same way. By putting those values into the same region we can hopefully remove the region when the program no longer needs that part of the term, while other parts are still needed. Alternatively, the term bound to a variable could be stored entirely in a region, which is not a good approach when only a part of the term becomes dead. Another finer-grained approach could be that every subterm of a term is stored in a separate region. This approach would mean that the number of physical regions needed to store a recursive term depends on the size of the term (which usually is not known at compile-time). Those regions could not be known at compile-time therefore it would be difficult to keep track and manipulate them.

3 Region Points-to Analysis

The goal of this analysis is to build, for each procedure, a region points-to graph that represents the partition of the memory used by the procedure into regions. The concept of a region points-to graph was introduced for Java in [2] and we adapted it in the context of Mercury. For a procedure p, a region points-to graph, $G = (N, E)$, consists of a set of nodes, N, representing regions and a set of directed edges, E, representing references between the regions. A node has an associated set of variables of p which are stored in the region corresponding to the node[2]. For a node n, its set of variables is denoted by $vars(n)$. The node n_X denotes the node such that $X \in vars(n_X)$. A directed edge is a triple. $(m, (f, i), n)$ denotes an edge between m and n, which is labelled by the type selector (f, i) and represents the structured relation between the variables in the two nodes. The type selector (f, i) selects the ith argument of the functor f [8].

The points-to graphs of *split* and *qsort* procedures are shown in Fig. 3. For *split* we see that the skeletons of the lists L and Ls are in the same region and that the elements are in the region pointed to by the edge with a label $([.], 1)$. Note that the edge with label a $([.], 2)$ is for the skeleton.

The region points-to graph, $G = (N, E)$, of a procedure p has to ensure the following invariants.

1. If a unification $X = f(\ldots, X_i, \ldots)$ (i.e., <= or =>) appears in p then $n_X, n_{X_i} \in N$ and there exists one and only one edge with a label (f, i) from n_X and $(n_X, (f, i), n_{X_i}) \in E$.
2. Every variable of p belongs to exactly one node and the variables in a node have the same type, which is regarded as the type of the node.

[2] From now on, we use the concepts of a node and the region corresponding to the node interchangeably.

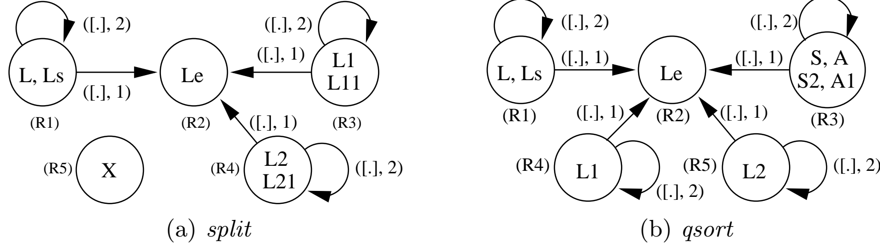


Fig. 3. The points-to graphs of *split* and *qsort*.

The task of the region points-to analysis then is to produce a graph that satisfies the invariants for each procedure in a program. The region points-to analysis is flow-insensitive, i.e., the execution order of the literals in a procedure does not matter, and consists of two analyses. One is an intraprocedural analysis that only deals with specialized unifications, ignoring procedure calls and the other is an interprocedural analysis that integrates the points-to graph of the called procedure (callee) with that of the calling procedure (caller) at each call site. The interprocedural analysis requires a fixpoint computation to calculate points-to graphs for recursive procedures. The points-to analysis applies the operations *unify* and *edge* to $G = (N, E)$ and updates G as follows.

- $\mathit{unify}(n, m)$: unify nodes n and m in the graph.
  - $N \leftarrow N \setminus \{n, m\} \cup \{k\}$, k is a new node and $\mathit{vars}(k) = \mathit{vars}(n) \cup \mathit{vars}(m)$.
  - $E \leftarrow E$ with the edges where all appearances of m, n are replaced by k.
- $\mathit{edge}(n, \mathit{sel}, m)$: create an edge with a label sel from node n to node m.
  - $G \leftarrow (N, E \cup \{(n, \mathit{sel}, m)\})$.

3.1 Intraprocedural Analysis

To specify this analysis, assume that we are analysing a procedure p with points-to graph $G = (N, E)$. The analysis works as follows.

1. Each variable in p is assigned to a separate node: for a variable X, n_X becomes a node in N and $\mathit{vars}(n_X) = \{X\}$.
2. The specialized unifications in p are processed one by one as follows:
   - An assignment $X := Y$: record that X and Y are in the same region because they point to the same memory block, i.e., $\mathit{unify}(n_X, n_Y)$.
   - A test $X == Y$: do nothing.
   - A deconstruction $X => f(X_1, \ldots, X_n)$ or a construction $X <= f(X_1, \ldots, X_n)$: create the references from n_X to each of $n_{X_1}, \ldots, n_{X_n}$ by adding the edges $\mathit{edge}(n_X, (f, 1), n_{X_1})$, …, $\mathit{edge}(n_X, (f, n), n_{X_n})$.
3. The rules in Fig. 4 are fired whenever applicable. Rules $P1$ and $P2$ are to ensure the first invariant. Rule $P3$ enforces the term representation for the variables of recursive types.

- $P1$:
  after *unify*(n, n'), k is the new node
  if
  $(k, sel, m), (k, sel, m') \in E \wedge$
  $m \neq m'$
  then
  unify(m, m')
- $P2$:
  after *edge*(n, sel, m)
  if
  $(n, sel, m') \in E \wedge\ m \neq m'$
  then
  unify(m, m')
- $P3$:
  after *edge*(n, sel, m)
  if
  $(k_1, m) \in E^+ \wedge$
  $(m, k_2) \in E^+ \wedge$
  $k_1 \neq k_2 \wedge$
  type$(k_1) =$ *type*(k_2)
  then
  unify(k_1, k_2)

  in which E^+ is the reflexive, transitive closure of E and *type*(n) returns the type of the node n.

Fig. 4. Intraprocedural analysis rules.

3.2 Interprocedural Analysis

The interprocedural analysis updates the region points-to graph of a procedure p by integrating the relevant parts of the points-to graphs of the called procedures into it. Assume that for a call $q(Y_1, \ldots, Y_n)$, the head of the defining procedure is $q(X_1, \ldots, X_n)$. The analysis is performed as follows.

1. Process each procedure call $q(Y_1, \ldots, Y_n)$ in p: integrate the graph of q, $G_q = (N_q, E_q)$, into the graph of p, $G_p = (N_p, E_p)$ by building the partial α mapping from N_q to N_p as follows:
   (a) Initializing the α mapping with $\alpha(n_{X_1}) = n_{Y_1}, \ldots, \alpha(n_{X_n}) = n_{Y_n}$. For those nodes n_{X_i}'s that have been unified in G_q, the corresponding nodes n_{Y_i}'s in the G_p are also unified. This is achieved by applying rule $P4$ in Fig. 5 to ensure that α is a function.
   (b) In the graph G_q, start from each n_{X_i}, follow each edge once and apply rules $P5$ - $P8$ in Fig. 5 when applicable. Those rules are to complete the α mapping and to copy the parts of G_q that are relevant to n_{X_i}'s into G_p. After some two nodes of G_p are unified (rules $P4$ and $P5$) or an edge is added to G_p (rules $P7$ and $P8$) we need to apply rules $P1$ or $P3$ ($P2$ is not applicable because its conditions cannot be satisfied) respectively to G_p in order to maintain the invariant that there is only one edge with a given selector between two nodes and to conform with the term representation for recursive types.
2. Step 1 is repeated until there is no change in G_p.

For the *qsort* example, the region points-to graphs of *split* and *qsort* after the points-to analysis reaches a fixpoint is exactly as shown in Fig. 3.

- $P4$:
  if
  $\alpha(n_{X_i}) = n_{Y_i} \wedge \alpha(n_{X_j}) = n_{Y_j} \wedge$
  $n_{X_i} = n_{X_j} \wedge n_{Y_i} \neq n_{Y_j}$
  then
  $unify(n_{Y_i}, n_{Y_j})$
- $P5$:
  if
  $(n_q, sel, m_q) \in E_q \wedge \alpha(n_q) = n_p \wedge$
  $(n_p, sel, m'_p) \in E_p \wedge$
  $\alpha(m_q) = m_p \neq m'_p$
  then
  $unify(m_p, m'_p)$
- $P6$:
  if
  $(n_q, sel, m_q) \in E_q \wedge \alpha(n_q) = n_p \wedge$
  $(n_p, sel, m_p) \in E_p \wedge$
  $\alpha(m_q)$ undefined
  then
  $\alpha(m_q) = m_p$
- $P7$:
  if
  $(n_q, sel, m_q) \in E_q \wedge$
  $\alpha(n_q) = n_p \wedge$
  $\nexists k : (n_p, sel, k) \in E_p \wedge$
  $\alpha(m_q) = m_p$
  then
  $edge(n_p, sel, m_p)$
- $P8$:
  if
  $(n_q, sel, m_q) \in E_q \wedge$
  $\alpha(n_q) = n_p \wedge$
  $\nexists k : (n_p, sel, k) \in E_p \wedge$
  $\alpha(m_q)$ undefined $\wedge$
  then
  m_p: a new node in G_p,
  $\alpha(m_q) = m_p$,
  $edge(n_p, sel, m_p)$

Fig. 5. Interprocedural analysis rules.

4 Live Region Analysis

The goal of live region analysis is to detect live regions at each program point and to decide which regions are created and removed by each procedure.

With every literal in the body of a procedure p, a *program point* is associated. An *execution path* in p is a sequence of program points, such that at runtime the literals associated with these program points are performed in sequence. We use the notions before and after a program point. Before a program point means the associated literal has not been executed, while after a program point means when its literal has just completed. The set of live regions (LR) at a program point is computed via the set of live variables (LV) at the program point.

4.1 Live Variables at a Program Point

A variable is live after a program point in p if:

- There exists an execution path in p containing the program point that instantiates the variable before or at the program point and uses it after the program point,
- OR it is an output variable of p, which is instantiated before or at the program point.

If we call $pre_inst(i, P)$ the set of variables instantiated before the program point i in the execution path P, $post_use(i, P)$ the set of variables used after i in P, $out(i)$ the set of variables instantiated by the goal at i, $out(p)$ the set of output variables of a procedure p then the set of live variables after i is:

$$LV_{after}(i) = \{V \mid \exists P : V \in (pre_inst(i, P) \cup out(i)) \cap (out(p) \cup post_use(i, P))\}.$$

If we call $in(i)$ the set of input variables to the literal at i, the set of live variables before i is:

$$LV_{before}(i) = (LV_{after}(i) \setminus out(i)) \cup in(i).$$

The LV_{before} of the first program point of an execution path of a procedure p is defined to be $in(p)$, the set of input variables of the procedure. The LV_{after} of the last program point is defined to be $out(p)$.

4.2 Live Regions at a Program Point

A region is live at a program point if it is reachable from a live variable at the program point. The set of regions that are reachable from a variable is defined:

$$Reach(X) = \{n_X\} \cup \{m \mid \exists (n_X, m) \in E^*(X)\},$$

in which $E^*(X)$ is defined:

$$E^*(X) = \{(n_X, n_i) \mid \exists (n_X, sel_0, n_1), \ldots, (n_{i-1}, sel_{i-1}, n_i) \in E\}.$$

The LR sets before and after a program point i are defined:

$$LR_{before}(i) = \bigcup (Reach(X)) \ \forall X \in LV_{before}(i).$$
$$LR_{after}(i) = \bigcup (Reach(X)) \ \forall X \in LV_{after}(i).$$

4.3 The Analysis

This analysis computes for each procedure the LR sets before and after each program point and the set of regions that the procedure may create, called *bornR* and the set of regions that it may remove, called *deadR*.

First, LR sets can be computed in two passes. The first pass is a syntax-based analysis to compute the LV sets for each program point. Note that because of mode correctness the LV sets are specific for a program point, independent of execution paths containing it. The second pass computes the LR sets for each program point from the corresponding LV sets and the region points-to graph.

After that, to reason about the lifetime of regions across procedure boundary, for a procedure p with its region points-to graph $G = (N, E)$, we define:

- $inputR(p)$ is the set of regions reachable from input variables.
- $outputR(p)$ is the set of regions reachable from output variables.
- $bornR(p)$ is the set of output regions that the procedure or any of the procedures it calls may create. Initially $bornR(p) = outputR(p) \setminus inputR(p)$.

- $deadR(p)$ is the set of input regions that the procedure or any of the procedures it calls may remove. Initially $deadR(p) = inputR(p) \setminus outputR(p)$.
- $localR(p) = N \setminus inputR(p) \setminus outputR(p)$.

The analysis then follows each execution path of a procedure p and applies the rules in Fig. 6 to any call q to update the *deadR* and *bornR* sets of q. Essentially, a region is removed from $deadR(q)$ if it needs to be live after the call to q in p; or if it is going to be removed more than once by q. A region is removed from $bornR(q)$ if the region is already live before the call to q; or if q is going to create the region more than once. When there is a change to those sets of q, q needs to be analysed to propagate the change to its called procedures. Therefore, this analysis requires a fixpoint computation.

- $L1$:

  if
  $r \in LR_{before}(pp(l)) \wedge$
  $r \in LR_{after}(pp(l)) \wedge$
  $r = \alpha(r') \wedge\ r' \in deadR(q)$
  then
  $deadR(q) = deadR(q) \setminus \{r'\}$

- $L2$:

  if
  $\alpha(r') = r \wedge \alpha(r'') = r \wedge$
  $r' \neq r'' \wedge$
  $r' \in deadR(q)$
  then
  $deadR(q) = deadR(q) \setminus \{r'\}$

- $L3$:

  if
  $r \in LR_{before}(pp(l)) \wedge$
  $r = \alpha(r') \wedge$
  $r' \in bornR(q)$
  then
  $bornR(q) = bornR(q) \setminus \{r'\}$

- $L4$:

  if
  $\alpha(r') = r \wedge \alpha(r'') = r \wedge$
  $r' \neq r'' \wedge$
  $r' \in bornR(q)$
  then
  $bornR(q) = bornR(q) \setminus \{r'\}$

Fig. 6. Live region analysis rules.

In the *qsort* program, *split* has three execution paths $\langle(1),(2),(3)\rangle$, $\langle(4),(5),(6),(7)\rangle$, and $\langle(4),(8),(9)\rangle$; *qsort* has two execution paths $\langle(1),(2)\rangle$ and $\langle(3),(4),(5),(6),(7)\rangle$. The LV and LR sets of *split* are in Tab. 1, of *qsort* in Tab. 2. Their corresponding *deadR* and *bornR* sets are: $deadR(split) = \{R1\}$, $bornR(split) = \{R3, R4\}$; $deadR(qsort) = \{R1\}$, $bornR(qsort) = \phi$.

pp	LV_{before}	LV_{after}	LR_{before}	LR_{after}
(1)	$\{X, L\}$	$\{\}$	$\{R5, R1, R2\}$	$\{\}$
(2)	$\{\}$	$\{L1\}$	$\{\}$	$\{R3, R2\}$
(3)	$\{L1\}$	$\{L1, L2\}$	$\{R3, R2\}$	$\{R3, R2, R4\}$
(4)	$\{X, L\}$	$\{X, Le, Ls\}$	$\{R5, R1, R2\}$	$\{R5, R2, R1\}$
(5)	$\{X, Le, Ls\}$	$\{X, Le, Ls\}$	$\{R5, R2, R1\}$	$\{R5, R2, R1\}$
(6)	$\{X, Le, Ls\}$	$\{L2, Le, L11\}$	$\{R5, R2, R1\}$	$\{R4, R2, R3\}$
(7)	$\{L2, Le, L11\}$	$\{L1, L2\}$	$\{R4, R2, R3\}$	$\{R3, R2, R4\}$
(8)	$\{X, Le, Ls\}$	$\{L1, Le, L21\}$	$\{R5, R2, R1\}$	$\{R4, R2, R3\}$
(9)	$\{L1, Le, L21\}$	$\{L1, L2\}$	$\{R4, R2, R3\}$	$\{R3, R2, R4\}$

Table 1. Live variable and live region sets of *split*.

pp	LV_{before}	LV_{after}	LR_{before}	LR_{after}
(1)	$\{L, A\}$	$\{A\}$	$\{R1, R2, R3\}$	$\{R3, R2\}$
(2)	$\{A\}$	$\{S\}$	$\{R3, R2\}$	$\{R3, R2\}$
(3)	$\{L, A\}$	$\{A, Le, Ls\}$	$\{R1, R2, R3\}$	$\{R3, R2, R1\}$
(4)	$\{A, Le, Ls\}$	$\{A, Le, L1, L2\}$	$\{R3, R2, R1\}$	$\{R3, R2, R4, R5\}$
(5)	$\{A, Le, L1, L2\}$	$\{Le, L1, S2\}$	$\{R3, R2, R4, R5\}$	$\{R2, R4, R3\}$
(6)	$\{Le, L1, S2\}$	$\{L1, A1\}$	$\{R2, R4, R3\}$	$\{R4, R2, R3\}$
(7)	$\{L1, A1\}$	$\{S\}$	$\{R4, R2, R3\}$	$\{R3, R2\}$

Table 2. Live variable and live region sets of *qsort*.

5 Program Transformation

The goal of program transformation is to introduce *create* and *remove* instructions based on the region liveness information. Each procedure is transformed by following its execution paths and applying the transformation rules in Fig. 7 to each program point. Assuming that we are analysing a procedure p. Let l_i be the associated literal at a program point i in p. A literal can be either a specialized unification denoted by *unif* or a call (user-defined or built-ins). We assume that all the specialized unifications as well as the built-ins do not remove or create any regions.

A region is created when it first becomes live, namely when the region is not live before i but is live after i. If l_i creates the region[3], then no annotation is added at i. Otherwise the region is created either by a caller of p or by p itself. The former means that the region does not need to be created again in p and no annotation is added at i. The latter occurs when the region belongs to either $bornR(p)$ or $localR(p)$ and a *create* instruction is added before l_i.

A region is removed when it ceases to be live. The first situation is when the region is live before i but not live after i. If l_i removes the region, then no *remove* instruction needs to be inserted in p. Otherwise if the region is removed

[3] When we say a region is created (removed) by a procedure it means that the region is either created (removed) by the procedure itself or by one of the procedures that it calls.

by p itself, namely the region is in either *deadR*, *localR*, or *bornR* sets of p, a *remove* instruction is inserted after l_i to remove the region. While the reason for removing the region when it belongs to the first two sets is straightforward, the removal of a region in $bornR(p)$ is allowed because it is acceptable for p to remove the region after i and re-create it later on (which p will fulfill somehow because the region is in $bornR(p)$). If p does not remove the region either then the region is removed by a caller of p and no annotation is introduced at i. The second situation is when the region is live after i, but not live before some program point j following i in a certain execution path. This can happen when i is a shared point among different execution paths and the region is live after i due to an execution path to which j does not belong. A *remove* instruction is added before l_j to remove the region.

- $T1:$

  if
  $l_i \equiv q(\ldots)\wedge$
  $r \in LR_{after}(i) \setminus LR_{before}(i)\wedge$
  $r \in (localR(p) \cup bornR(p))\wedge$
  $r = \alpha(r')\wedge\ r' \notin bornR(q)$
  then
  add "*create r*" before l

- $T2:$

  if
  $l_i \equiv X <= f(\ldots)\wedge$
  $r \in Reach(X) \setminus LR_{before}(i)\wedge$
  $r \in LR_{after}(i)\wedge$
  $r \in localR(p) \cup bornR(p)\wedge$
  then
  add "*create r*" before l

- $T3:$

  if
  $l_i \equiv q(\ldots)\wedge$
  $r \in LR_{before}(i) \setminus LR_{after}(i)\wedge$
  $r \in localR(p) \cup deadR(p) \cup bornR(p)$
  $\nexists r' : r = \alpha(r') \wedge r' \notin deadR(q)$
  then
  add "*remove r*" after l

- $T4:$

  if
  $l_i \equiv unif\wedge$
  $r \in LR_{before}(i) \setminus LR_{after}(i)\wedge$
  $r \in localR(p) \cup deadR(p) \cup bornR(p)$
  then
  add "*remove r*" after l

- $T5:$

  if
  l_j is right after l_i in an execution path
  $r \in LR_{after}(i) \setminus LR_{before}(j)\wedge$
  $r \in localR(p) \cup deadR(p) \cup bornR(p)$
  then
  add "*remove r*" before l_j

Fig. 7. Transformation rules.

The result of the program transformation of the *qsort* program has been shown in Fig. 1.

6 Prototype Implementation and Discussion

We implemented a prototype of the algorithm as a source-to-source transformation in the MMC version 0.12.0. The analyser operates as one of the last analyses on the High Level Data Structure representation of the original source code and produces human-readable source code with region support. To realise Mercury with RBMM the runtime system of Mercury should be extended with support for regions, which is out of the scope of this paper.

Our region analysis and transformation work correctly for deterministic programs in which the condition of if-then-else constructs, if any, can only be a goal that does not require creating or removing any regions inside itself. Note that such a deterministic program can still have different execution paths. There are two reasons for these restrictions. Firstly, the region liveness analysis only takes into account forward execution. Therefore the removal and creation of regions are only correct w.r.t. forward execution as we have in deterministic programs. Secondly, even with a deterministic program, we need to be sure that when an execution path of the program is taken all the region instructions on that path are actually executed. A *canfail* goal could cause problems such as created regions would never be removed or a region would be created or removed more than once, and so on. In deterministic programs such *canfail* goals can only appear in the condition of if-then-elses. Therefore, the condition goal of if-then-elses is restricted so that no region instructions occur in this nondeterministic context.

Our approach is a valid starting point for RBMM for general Mercury programs. The authors in [5,7] describe an enhanced runtime for Prolog with RBMM, which provide the support for nondeterministic goals. The idea of that enhanced runtime is that backtracking is made transparent to the algorithm for deterministic programs by providing a mechanism to undo changes to the heap memory, restoring the heap memory to the previous state at the point to which the program backtracks. We believe that with a similar runtime support our algorithm can be used unchanged to support full Mercury programs. In future work we will investigate whether the runtime support can be lessened by exploiting the determinism information available in Mercury.

To make it possible to study the programs with region support we implemented a region simulator in which *create* and *remove* instructions are valid Mercury predicates that record each time a region is created or removed. The simulator is not fully automated in the sense that a program annotated with the two predicates generated by our prototype needs to be manually modified so that it can be compiled and executed as a normal Mercury program. By using the simulator we collected the information about the total number of regions created and the maximum number of regions during a program execution. We can also verify that all regions created by the program are also removed when the program finishes. Unfortunately, the sizes of the regions, which are essential

to measure memory saving, are not collected by the simulator. Collecting those figures as well as the runtime performance of the transformed programs would require a working system of Mercury with support for RBMM.

	nrev	**qsort**	**dnamatch**	**argo_cnters**	**life**
Total number of regions	500	1001	1448	2000	1216
Maximum number of regions	4	12	17	34	12
Analysis time (sec)	0.010	0.016	0.280	1.679	0.189
Normal compilation time (sec)	0.259	0.233	0.309	0.66	0.340

Table 3. Experimental results.

The results in Tab. 3 show that the total number of regions is always much larger than the maximum number of regions. It means that there are quite many regions having a short lifetime and the programs are able to remove them. The analysis time[4] ranges from 3.8% to 254% of the normal compilation time[5], which is acceptable for a prototype implementation of the algorithm. The analysis time for *argo_cnters* is large likely due to the fact that this benchmark processes the functors of a large arity (20), creating a graph with many nodes and edges, and that the behaviour of the program itself causes many nodes to be unified.

Up to our knowledge, the pioneering and only attempt to apply RBMM to a standard LP system is the work of Makholm and Sagonas [7] for XSB Prolog. Their work focused on the WAM runtime extensions required for regions, not the region analysis. In that aspect, they did validate that RBMM can work well with nondeterministic code, a unique feature of logic programming. Our work has concentrated on the static analysis. In our experiment we use the benchmark programs *nrev*, *qsort*, and *dnamatch*, also used in [7]. Our algorithm produces better analysis results for the first two programs, as it distributes the list skeletons and the elements in different regions and detects a shorter lifetime of regions, which would imply less memory consumption at runtime. Our RBMM *qsort* and *nrev* programs use no more memory than what is needed to store the input list. While in [7] *nrev* used maximally double and *qsort* 1.66 times the memory size of the input list. For *dnamatch* our analysis has the same problem as reported in [7], when temporary data is put in the same regions as the input and output data. This problem also happens for *agro_cnters* and *life*, which should cause poor performance of RBMM.

In our analysis, the problem is due to the imprecision of the region points-to analysis. The current region points-to analysis cannot capture the fact that two regions are the same in one execution path but not in another and just always unifies them. If they would have been kept separate, one of them had been removed when only the other is live. This problem has also been reported

[4] The analyses are executed on a Pentium 4 2.8MHz with 512MB RAM running Debian GNU/Linux 3.1 machine, under a usual load.

[5] The compilation time using the MMC 0.12.0 with default compilation options.

in the RBMM research for functional programming [6,4] with many solutions such as *storage mode analysis*, and those in [1,4] but as far as we understand those solutions have not solved the problem completely. A tentative approach to tackle this problem is to enhance the region points-to analysis so that it is able to detect which variables are definitely in the same regions and which ones are possibly in the same regions. We will investigate this possibility in future work.

7 Conclusion

We have developed and implemented an algorithm for region analysis and transformation for deterministic Mercury programs. While the work in [7] emphasised on the extension of the Prolog runtime system for supporting regions, our work here focuses on the static detection of regions and the introducing region annotations, exploiting the available type and mode information of Mercury. Experimental results of using a prototype implementation of the algorithm are reported for several programs. We obtained promising results for some programs, while for some others the typical shortcomings known from the functional programming world show up. In particular, we now have a system in which the combination of CTGC and RBMM can be investigated. The correctness proofs of the analysis and transformation will be topics for further work. Future work also includes the modular region analysis and the improvement of the precision of the region points-to graph and analysis.

References

1. A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, 1995.
2. S. Cherem and R. Rugina. Region analysis and transformation for Java. In *Proceedings of the 4th International Symposium on Memory Management*, pages 85–96. ACM Press., October 2004.
3. D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.*, pages 282–293. ACM Press., 2002.
4. F. Henglein, H Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming.*, pages 175–186. ACM Press., 2001.
5. H. Makholm. A region-based memory manager for Prolog. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 25–34. ACM Press., 2000.
6. H. Makholm. Region-based memory management in Prolog. Master's thesis, University of Copenhagen, 2000.
7. H. Makholm and K. Sagonas. On enabling the WAM with region support. In *Proceedings of the 18th International Conference on Logic Programming*. Springer Verlag., 2002.

8. N. Mazur. *Compile-time garbage collection for the declarative language Mercury.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, May 2004.
9. N. Mazur, G. Janssens, and M. Bruynooghe. A module based analysis for memory reuse in Mercury. In *Proceedings of Computational Logic - CL 2000, First International Conference*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269. Springer-Verlag, 2000.
10. N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for Mercury. In *Proceedings of the 17th International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2001.
11. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming*, 29(1-3):17–64, October-December 1996.
12. M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:245–265, 2004.
13. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation.*, 132(2):109–176, February 1997.

Delay and events in the TOAM and the WAM

Bart Demoen* and Phuong-Lan Nguyen†

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

† Institut de Mathématiques Appliquées, UCO, Angers, France
nguyen@uco.fr

Abstract. Ten years ago, published experiments comparing two approaches to implement delayed goals seemed to indicated a clear edge for the TOAM approach (in B-Prolog) over the WAM approach (in SICStus Prolog). The old experiments are redone and a better explanation is offered for the performance difference. hProlog (a WAM-based Prolog implementation) is used to show that with some effort, the traditional WAM approach to freeze/2 performs similar to the TOAM approach. We do the same for the event mechanism of B-Prolog.

1 Introduction

We must assume working knowledge of Prolog and its implementation. For a good introduction to Prolog see [4]; to the WAM, see [1,13]; for B-Prolog and the TOAM, see [14]; the SICStus Prolog implementation is described in [3]. hProlog is the successor of dProlog [6]; it is available from the first author. We have used the following releases of these systems: B-Prolog 6.9-b4.2 in Section 8 and 6.9-b3.2 elsewhere, SICStus Prolog 3.12.0, hProlog 2.6.*. The benchmarks were all run on a Pentium, 1.8GHz, under Debian, and while running the benchmarks, garbage collection was disabled by starting the system with enough initial memory, whenever possible. We also used SWI-Prolog Version 5.5.40 and Yap-4.5.7 in some of the experiments.

The predicate *freeze/2* has its origin in Prolog II as *geler/2*. [2] is the first publication of its implementation in the context of the WAM. Later implementations use meta-structures [10] or attributed variables [8]. More recently the TOAM [15] introduced the concept of delay clauses which can be used for the specification (at the source level) of freeze/2. Delay clauses have since then been replaced by *Action Rules*, but (AFAWK) the basic implementation principle - by suspension frames on the stack - has remained the same. A short introduction to suspension frames is given in Section 2.

The issue of putting asleep and waking up goals is not restricted to freeze/2: the efficiency of constraint solvers depends crucially on the efficiency of the mechanism to put goals asleep (conditionally) and waking up those goals when certain events occur. Instantiation is just one such event and it is related to the Herbrand solver. Other solvers have their own events, e.g. *upper bound changed* for a finite domain solver. However, when the other solver is embedded in an

untyped context like Prolog, there is a huge difference between the Herbrand event of instantiation, and events from the other solver: Herbrand instantiation of any variable (any in the sense of *possibly belonging to a non-Herbrand solver*) is supposed to be supported and the underlying Prolog system must implement (a form of) freeze/2 for that. The other (non-Herbrand) events can easily be implemented **without** freeze/2 by using *passive* attributed variables to which any information can be attached and which do not react necessarily to Herbrand instantiation. Whether this difference is necessarily important at the implementation level is a different issue, but it should be clear that the wakeup of a goal because an upper bound changed, has nothing to do with Prolog's unification routine.

In the 1996 paper [15], the author explains the use of delay clauses (now named action rules) for implementing delay - as done in B-Prolog. An experimental comparison is presented there as well: there are only a few benchmarks and the comparison is with SICStus Prolog. In other sciences (experimental physics, biology ...) it is quite common to redo experiments. In computer science however, it seems rather uncommon: experiments are seldom redone. However, changes in hardware and software affect the outcome of experiments over time, and more importantly new explanations of experiments could be discovered by redoing them and monitoring them in different ways. The experiments in [15] have never convinced us, so we devised the following plan: (1) redo the experiments of [15] and check whether the conclusions put forward there are (still) valid; (2) use hProlog as a lab environment for checking whether the conclusions put forward in [15] are valid in another WAM based Prolog implementation, and in particular whether a traditional WAM approach to delay is *necessarily* less efficient. We will also study the *event* mechanism of B-Prolog which is implemented in the TOAM with the same stack based mechanism as freeze, and see how a heap based approach can be made to perform as good. We will also use hProlog for events.

Apart from some dissatisfaction with the experiments in [15], our motivation for this work also derives from the CHR implementation context. (1) CHR implementations rely on a freeze-like mechanism to trigger postponed rules (see for instance [9]): we were interested in improving the performance of that aspect of the implementation within the K.U.Leuven CHR system (see `http://www.cs.kuleuven.ac.be/~toms/Research/CHR/`; (2) more recently, we became involved in the design of a compilation schema of CHR to action rules that uses heavily the B-Prolog event mechanism [11]; anticipating that this might turn out a powerful translation schema, we wanted to find out how efficient a WAM-based implementation of this event mechanism could fare.

In Section 3 we reproduce the timings and conclusions from [15]. The outline of the rest of the paper can be found at the end of that section. We start with a short introduction to suspension frames.

2 On suspension frames.

The suspension frame mechanism of the TOAM has been described in [15] in much detail: there is no point in repeating it here. Instead, we offer an alternative view to suspension frames, namely one that introduces them in the more widely known context of the WAM. First we give a quick review of how freeze/2 can be implemented in the WAM with attributed variables. For the sake of our explanation, the goal will be ?- freeze(X,p(A,B,C)), X = 1. and the predicate p/3 has only one clause: p(A,B,C) :- <Body>. where <Body> stands for an arbitrary body.

The WAM approach: the term p(A,B,C) is put on the heap and the variable X gets an attribute that will make sure the goal p(A,B,C) is called when X is instantiated; when X is unified with 1, the arguments of the heap term p(A,B,C) are put in the first three argument registers, and the clause for p/3 is entered. If p/3 needs an environment, it is constructed at this point only. We name this the meta-call approach.

Suspension frames in the WAM: think of a WAM implementation without last call optimization. Construct the new clause
newp(A,B,C,Susp) :- reentry(Susp), <Body>. where reentry/1 has a double function: (at compile time) it forces newp/4 to have an environment on the stack; and when reentry is executed, it first unifies Susp with the data structure susp(<E>,<Ad>) and then returns to the caller. <E> is the address of the current environment[1], and <Ad> is the start address of the code of <Body>. Finally, replace the goal freeze(X,p(A,B,C)) by

```
(var(X) ->
      p(A,B,C)
;
      newp(A,B,C,Susp),
      attach_attr(X,Susp)
)
```

When X is unified with 1, X's attribute tells where to jump to and which environment to use. We need to take care of returning correctly from newp/4, but that is easy as long as a suspension frame can not be reactivated before the end of the newp/4 clause.

Advantage of the suspension frame on activating the delayed goal: on activating the woken goal, the meta-call approach must move all the arguments to the appropriate registers and create the environment for the entered clause; the suspension frame approach needs to install only some abstract machine pointers and can start executing the body of the clause immediately: the frame is already there. This could be important when the same delayed goal is woken several times: this can happen when the same goal is waiting for the instantiation of more than one variable.

[1] Now you know why we got rid of LCO

Disadvantages of the suspension frame approach: one needs a garbage collector for the environment stack, because unreachable suspension frames can become trapped. Also, there will be more argument saving (and restoring) in the environment in newp/4 than in p/3.

The above describes the basic concept of suspension frames on the stack: low level support can be provided and LCO can be retrieved to some extent. The whole idea fits the TOAM rather well, because the TOAM passes arguments through the stack. Note that with suitable modifications, the suspension frames could be kept on the heap as well.

We can now also give a partial introduction to action rules, in as far as they are used for freeze/2: the above query and goal would be written as follows with action rules

```
p(A,B,C,X), var(X), {ins(X)} => true.
p(A,B,C,Y) => <Body>.

?- p(A,B,C,X), X = 1.
```

The first goal in the query has the same effect as freezing the execution of `<Body>` until X is instantiated: the first p/4 clause creates a suspension frame. The second clause encapsulates the action on instantiation of X.

3 The experiments 10 years ago

We start here by quoting the ratios of timings reported in [15] and a verbatim quote of its conclusion: we have left out the absolute times and figures related to SICStus Prolog native code and to memory usage.

	block	freeze
nreverse	8.37	28.67
queens	2.22	13.87
sendmory	1.96	10.71
psort	2.31	13.26

Table 1. Execution time ratio SICStus/B-Prolog in 1996

1. *While using freeze and delay clauses does not cause much difference in execution time in B-Prolog, using block declaration is much faster than using freeze/2 in SICStus Prolog.*
2. *For the four programs, B-Prolog is significantly faster than SP-bc*[2] *[and . . .]. The speed-ups are due mostly to the novel implementation method of delay adopted in B-Prolog. For the original nreverse program without delay, B-Prolog is only 45 percent faster than SP-bc.*

[2] SICStus Prolog byte code - the emulator

3. *It is difficult to tell to what extent delay affects the execution time because to do so we have to get rid of the time taken to run predicates that never delay. However, as more than 90 percent of the predicate calls in nreverse delay in execution, the ratios in the row for nreverse roughly tell us about the difference between the performance of the two systems.*

The first conclusion follows objectively from the reported figures - the only criticism could be that only four benchmarks are used. However, [15] fails to investigate reasons for the difference between freeze/2 and block declarations in SICStus Prolog: we will get into this in Section 5.

The second conclusion attributes the speed-up to the novel implementation method of delay. This conclusion does not follow from the figures: it is an interpretation, or a possible explanation, and a scientific approach consists in putting this conclusion to scrutiny.

The last sentence in the third conclusion seems to mean that the basic performance of freeze (respectively block declaration) compared to the performance of delay in B-Prolog, is about 28 (respectively 8); we will come back to this in Section 4 after we have redone the experiments. Section 5 finally uncovers the truth about the comparison with SICStus Prolog. From Section 6 on, we use hProlog, initially on the original benchmarks and in Section 7 on a set of artificial benchmarks which is used in order to understand better the performance of the different aspects of freeze/2. Section 8 deals with the B-Prolog *events*.

4 The same experiments 10 years later

The benchmarks used in 1996 are available in the distribution of B-Prolog. It is a simple matter to rerun them, but since machines have become faster, it was necessary to run the benchmarks with larger input: for queens, it used to be 8 and now it is 10; for nreverse, it was 100 and now 500; for psort, it was a list of length 15 and now 19. The results are in Table 2.

	block	freeze
nreverse	2.00	4.00
queens	2.26	14.10
sendmory	1.88	12.87
psort	1.84	18.67

Table 2. Execution time ratio SICStus/B-Prolog in 2006

The 1996 conclusions haven't changed much in 2006:

1. Block declarations are still much faster than freeze/2 in SICStus Prolog.
2. The figures for nreverse have improved a lot for SICStus Prolog, while the other figures got a little worse: on the whole B-Prolog is still significantly faster than SICStus Prolog. The reasons are still unclear.

3. If the reasoning for 1996 conclusion 3 is correct (the result of nreverse is indicative for the difference in implementation technology), one must conclude that in 2006 the TOAM implementation of delay is only 2 (for block) to 4 (freeze) times faster than the SICStus technology. That is an improvement for SICStus Prolog of a factor of 4 to 7.

Table 2 is remarkable in two respects:

- the figures in the block column are all close to two
- the freeze figure for nreverse is quite different from the other freeze figures which in turn are rather close

The first point seems to indicate that **two** is a fundamental constant for the ratio between SICStus block and TOAM freeze.

The second point suggests that the nature of the benchmarks themselves should explain the outcome. The nreverse benchmark is the only deterministic benchmark, while the other three benchmarks find their solution(s) by backtracking during a labeling phase (queens, sendmory) or by generating permutations (psort). This means that while in nreverse each delayed goal is activated once, the delayed goals in the other three benchmarks are activated many times. A hasty conclusion would be that backtracking accounts for the qualitative difference in the 2006 figures for nreverse and the other three. Section 5 shows that the explanation is very different, but we need to dig deeper to find the truth.

5 The difference between freeze/2 and block declarations

In SICStus Prolog, freeze/2 is implemented in intrins1.pl basically as:

```
:- block freeze(-, ?).
freeze(_, Goal) :- call(Goal).
```

The block declaration makes sure that the body is delayed until the arguments marked with - are free.

Inter-module specialization (or a special purpose hack) could easily transform a goal like `freeze(X,foo(A,B,X,D,E))` to `foo_blocked(A,B,X,D,E)` and add the following definition for *foo_blocked/5*

```
:- block foo_blocked(?,?,-,?,?).
foo_blocked(A,B,C,D,E) :- foo(A,B,C,D,E).
```

We name this specialization *block introduction*. SICStus Prolog does not do it. Since a block declaration is compiled to specialized instructions, block introduction can lead to great performance gains. In fact, the absence of block introduction is the only reason for the huge difference between block and freeze for the nreverse benchmark.

B-Prolog on the other hand applies a similar specialisation for calls to freeze/2: it could be named *action rule introduction*. This explains why there is virtually no difference between the use of action rules and freeze/2 in B-Prolog.

We are still left with the question: why is the ratio freeze/block so much bigger for the three backtracking benchmarks than for nreverse ? The answer is: *nested freezes*. The psort benchmark contains the goal `freeze(X,freeze(Y,X=<Y))`. The inner nested freeze/2 goal is executed at the moment that X becomes instantiated. This means that the query

```
?- L = [1,2,3,4,5], freeze(X,freeze(Y,X=<Y)), member(X,L), fail.
```

executes 6 times a call to freeze/2, 5 of which by a meta call.

One can get rid of the nesting of two freezes as follows: the goal `freeze(X,freeze(Y,goal(X,Y))` is transformed to the conjunction `NewGoal = newgoal(X,Y,_), freeze(X,NewGoal), freeze(Y,NewGoal)` with newgoal/2 defined as:

```
newgoal(X,Y,New) :-
        (nonvar(X), nonvar(Y), var(New) ->
              New = 1,
              goal(X,Y)
        ;
              true
        ).
```

We name this transformation *unnesting*: only when both X and Y are instantiated is `goal(X,Y)` executed and the variable New makes sure that it is executed at most once. The above query executes only two freeze goals when the nested freezes are unnested. The effect of unnesting the psort benchmark is very pronounced: the number of calls to freeze/2 goes down from 2621858 to 38. The sendmory and queens benchmarks contain similarly nested freezes. Unnesting has a huge effect on the three involved benchmarks. Table 3 adds one column to Table 2: the ratio for the unnested freeze programs in SICStus Prolog to the original program in B-Prolog.

	block	freeze	unnest
nreverse	2.00	4.00	4.00
queens	2.26	14.10	4.61
sendmory	1.88	12.87	3.91
psort	1.84	18.67	3.71

Table 3. Execution time ratio SICStus/B-Prolog in 2006: with unnested freeze

Conclusion 1 from 1996 can now be completed with an explanation:

> *The huge difference between block and freeze for queens, sendmory and psort is caused by not unnesting nested freezes.*

Note that the ratio between the unnested freeze and block performance is now almost constant and equal to two. Combining unnesting with block introduction

erases the difference completely of course. An example shows how this works: the goal `freeze(X,freeze(Y,foo(Z))` is transformed to `foo(X,Y,Z)` and foo/3 is defined as

```
:- block foo(-,?,?), foo(?,-,?).
foo(_,_,Z) :- foo(Z).
```

with some obvious optimizations when X and/or Y is in the term Z syntactically.

B-Prolog performs unnesting. One can verify this by inspecting directly the generated abstract machine code, or by performing an indirect experiment in which the manifest nesting is hidden, i.e. by transforming the goal `freeze(X,freeze(Y,goal(X,Y))` to `freeze(X,myfreeze(Y,goal(X,Y))` and defining

```
myfreeze(X,Y) :- freeze(X,Y).
```

This transformation reduces the performance of psort in B-Prolog by a factor of 1.9. This shows important unnesting is for performance.

First new conclusions related to the old experiments Since calls to freeze/2 are almost always manifest - i.e. at the source level, the second argument is not a variable - block introduction is almost always applicable and it does away easily with the performance difference (between block and freeze for SICStus Prolog) for all the benchmarks. When block is not available, unnesting reduces the difference between a WAM freeze implementation and a TOAM one.

The ratio B-Prolog/block is close to two for all four benchmarks. Given that B-Prolog is about 50% faster than SICStus Prolog for plain Prolog code, this could indicate that there is a performance edge in action rules: by now, we tend to be cautious about such a conclusion.

6 Digging deeper ...

Starting from the experience of block versus freeze, we have looked more closely at the effect of the compilation and execution process itself on the performance. From now on, we will use hProlog for comparison with B-Prolog because it is easier for us to perform implementation experiments with hProlog than with SICStus Prolog. Still, we believe that the results we have obtained in the hProlog context are of value in any WAM implementation. We first show in Table 4 the benchmark results for B-Prolog and hProlog without any changes to the benchmarks or to the hProlog implementation. hProlog has freeze/2, but no block declarations: freeze/2 is build on top of (dynamic) attributed variables [5].

From the figures in Table 4, one would naively conclude that the TOAM way is superior to the WAM way. However ...

– The hProlog implementation of attributed variables [5] uses one slot (a heap word) for **all** attributes: in general this slot contains a pointer to a list

	B-Prolog	hProlog
nreverse	40	102
queens	212	311
sendmory	254	769
psort	1193	4447

Table 4. B-Prolog and hProlog initially

with pairs (module,attribute value). Freeze/2 is not treated in any special way and freeze/2 suffers because of this generality: (1) when two attributed variables are bound, the underlying C implementation doesn't know about the special meaning of the freeze attribute and the freeze handler written in Prolog takes care of all actions; (2) when a frozen variable is instantiated, the general handler is called, while the frozen goal could be called directly. These general actions result in performance loss. If the implementation has a special freeze slot in the attributed variable, this overhead can be avoided to a large extent. We will refer to this as *spec_slot*. In the context of hProlog, we mimic it by using the unique attribute slot only for freeze/2. Low-level built-ins deal with this slot, but no further knowledge about freeze/2 is pushed deeper in the implementation. These low-level built-ins existed prior to working on this paper. In order to use the spec_slot freeze/2 version, the user simple imports the freeze library which then overrides the general implementation.

– Without special treatment by the compiler, the goal `freeze(X,Z is X+Y)` is treated exactly as the conjunction `Goal = Z is X+Y, freeze(X,Goal)`: the goal is constructed as a heap term and when X becomes instantiated, a meta-call of Goal is executed. No advantage is taken of the fact that the Goal is manifest. A meta-call cannot be avoided, but if a meta-call of arithmetic operations results in interpretation of the arithmetic, it is better to rewrite the manifest goal above as `freeze(X,newp(X,Y,Z))` and define the new predicate `newp(X,Y,Z) :- Z is X+Y`. We will refer to this as *fold*. The benchmarks sendmory and psort contain fold-able freezes - nreverse and queens do not.
– We have discussed previously unnesting: nreverse does not benefit from it.

Table 5 shows the benchmarks with unnesting, folding and special slot. The table does not show unnest and fold figures for B-Prolog: that would not be relevant to the point we want to make: B-Prolog performs unnesting and fold already and applying these transformations at the source level has a negative impact.

More new conclusions related to the old experiments The biggest gain clearly comes from using a dedicated slot for freeze/2. Also folding can have a

	nreverse	sendmory			psort			queens	
	orig	orig	fold	unnest	orig	fold	unnest	orig	unnest
B-Prolog	40	254			1193			212	
hProlog	102	796	453	439	4447	3366	2352	311	224
hProlog spec_slot	31	548	222	228	2206	1331	1105	175	176

Table 5.

big impact, but the extent of its effect seems hProlog specific: it is much smaller in SICStus Prolog.[3]

Our conclusion is that similar performance for delaying goals can be achieved in the TOAD and the WAM: it is a matter of using a dedicated freeze slot and some simple source-to-source transformations. The best figure for hProlog is always better than the B-Prolog figure. hProlog is also a bit faster than B-Prolog on ordinary Prolog code. In any case, from these benchmarks we can't conclude much about the basic mechanisms with which freeze is implemented. Artificial experiments can shed new light.

7 Artificial experiments

There are several actions related to a goal like `freeze(X,Goal)` in which X is free and (some time) after which X becomes instantiated: (1) the Goal is frozen on X; (2) the Goal is woken. Our artificial benchmarks are intended to measure the cost of these two actions separately: Tables 6 and 7 report on freezing, while Tables 8 and 9 report on wakeup. Two more variants interest us:

1. the same variable can be frozen several times, and later become instantiated, or one can freeze several variables and then instantiate them during one atomic unification; the former is reported on in Tables 6 and 8, while the latter is reported on in Tables 7 and 9
2. the arity of the goal to be frozen and later woken up is important, because both in the WAM and in the TOAM, there is code to be executed for each argument; the tables report timings for a goal with arity 1 and arity 4, separated by a /

We have always subtracted the time of dummy loops. The figures for Yap, SICStus Prolog and SWI-Prolog are only given so that the reader can place the B-Prolog and hProlog figures in a wider performance context.

Conclusions about the artificial experiments The following observations were most striking to us:

[3] The reasons do not fit in the margin, but are obvious and uninteresting for whomever studied the sources.

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	109 / 128	59 / 84	44 / 57	188 / 195	380 / 405	528 / 591
2 vars 1 goal	261 / 310	211 / 278	164 / 238	418 / 485	1100 / 1165	1055 / 1198
3 vars 1 goal	346 / 425	267 / 386	196 / 328	588 / 773	1791 / 1960	1568 / 1795
4 vars 1 goal	433 / 524	340 / 526	240 / 436	758 / 970	2500 / 2660	2113 / 2404

Table 6. Freezing: different variables on one goal each

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	109 / 128	59 / 84	44 / 57	188 / 195	380 / 405	528 / 591
1 var 2 goals	235 / 291	281 / 384	139 / 249	368 / 475	985 / 1071	927 / 1101
1 var 3 goals	328 / 418	421 / 545	214 / 323	528 / 668	1610 / 1735	1289 / 1557
1 var 4 goals	444 / 525	567 / 768	230 / 479	678 / 853	2237 / 2365	1601 / 2030

Table 7. Freezing: the same variable on one to four goals

- B-Prolog is **less** sensitive than any other system to the arity of the frozen goal when the freeze actually takes place; when the goal is woken, B-Prolog is amongst the systems **most** sensitive to the arity of the woken goal; this seems counterintuitive, and it took some time to find the explanation; space limitiations prevent us from going into this
- using the special slot in hProlog again really pays off: the gain is huge and hProlog-spec_slot is systematically the best of all

Apart from the special slot, one specialization also payed off quite well: when one atomic unification instantiates just one frozen variable (resp. two), the general handler is not called, but a specialized version which *knows* there is only one goal (resp. two) to call: this requires just a little extra C-code in the routine that installs the handler.

The most important conclusion seems to be that for the implementation of freeze/2

the WAM approach can compete favourably with the TOAM approach

It is also clear that it requires a careful implementation effort.

8 Events: multiple wakeups

Our benchmarks have dealt with only one solver event: Herbrand instantiation. The most distinguishing fact about Herbrand instantiation is that - in the absence of backtracking - it can happen only once for a particular variable, while other solver events (e.g. bounds changed) can happen many times for the same finite domain variable. Non Herbrand solver events are more important for constraint solvers and a comparison between WAM and TOAM along that line is in order. B-Prolog has another event mechanism that is not related to solvers in the traditional sense. We give just a small example with one action rule and a query:

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	244 / 253	695 / 687	173 / 172	1098 / 1110	2305 / 2355	1871 / 1883
2 vars 1 goal	539 / 585	1158 / 1165	360 / 361	2095 / 2145	4762 / 4848	3206 / 3243
3 vars 1 goal	806 / 948	1972 / 1982	646 / 655	3103 / 3180	7162 / 7257	4628 / 4683
4 vars 1 goal	1129 / 1332	2532 / 2556	781 / 787	4095 / 4200	9606 / 9700	6077 / 6195

Table 8. Waking up: different variables on one goal each

	B-Prolog	hProlog	hProlog-specslot	SICStus	SWI	Yap
1 var 1 goal	244 / 253	695 / 687	173 / 172	1098 / 1110	2305 / 2355	1871 / 1883
1 var 2 goals	432 / 575	809 / 804	280 / 284	1893 / 1955	3615 / 3687	2323 / 2343
1 var 3 goals	612 / 744	884 / 875	346 / 351	2753 / 2823	4307 / 4413	2645 / 2676
1 var 4 goals	803 / 985	976 / 981	436 / 435	3543 / 3638	4955 / 5087	3064 / 3089

Table 9. Waking up: the same variable with one to four goals

```
p(X), {event(X,M)} => write(M).
?- p(X), post_event(X,hello), post_event(X,world).
```

The query results in the output *helloworld*.

Roughly, every time there is a call to `post_event(X,M)` the bodies of the action rules that were called previously with X as the channel variable in the event, are executed.

8.1 Events in the WAM

It is not difficult to mimic the B-Prolog event behaviour in a Prolog system with attributed variables. We show this by an hProlog program (also working under SWI-Prolog) equivalent to the above example:

```
p(X) :- put_attr(X,event,p1(X)).
p1(X,M) :- write(M).
post_event(X,M) :-
        get_attr(X,event,Goal),
        call(Goal,M).
```

A more general translation schema goes as follows:

```
h(Z), {event(X,Message)} => Body.
```

is translated to:

```
h(Z) :-
    (get_attr(X,event,A) ->
       put_attr(X,event,[h1(Z)|A])
    ;
       put_attr(X,event,[h1(Z)])
```

```
post_event(X,Message) :-
    get_attr(X,event,Goals),
    call_goals(Goals,Message).

call_goals([],_).
```

```
   ).

h1(Z,Message) :- Body.
```

```
call_goals([G|Gs],Message) :-
    call_goals(Gs,Message),
    call(G,Message).
```

Two issues are not addressed by the above code:

– the matching (as opposed to unifying) semantics of a head in an action rule is not taken into account: hProlog does not (yet) have instructions for doing matching; however, this issue does not influence the benchmarks
– extra conditions in the action rule are not catered for: a small adaptation would; this issue does not influence the benchmarks either

8.2 Optimizing events in hProlog

The above code is not good enough to achieve performance comparable to what B-Prolog offers for its events. Here is an account of the extra implementation effort we needed:

– B-Prolog executes goals in the chronological order in which they where installed; this is achieved above by the left-recursion in call_goals/2; in the actual implementation, we call reverse/2 first and then call a tail recursive version of call_goals/2; that is more efficient; reverse/2 was implemented at a low level in hProlog
– for hProlog, we have used a new low level attribute built-in, which amounts to using a dedicated attribute slot for the events and which renders the body of h(Z) in the example above to a single call to a built-in predicate
– again referring to the example above, instead of adding the goal h1(Z) to the goal list, we add h1(_,Z): this allows us to replace the goal `call(G,Message)` by a call to a new built-in predicate `event_call(Message,G)` whose working is as follows: it calls a copy of G in which the first argument is replaced by Message[4]. In hProlog, it is relatively expensive to go from the functor N/A to a the code-entry address of the predicate with functor N/(A+1); this new approach avoids that cost
– the whole of call_goals/2 has been implemented as two new abstract machine instructions: we have done so in a totally different context before; see [12]

Clearly, we needed some effort in our WAM based implementation to make it compare favourably to the TOAM approach, but not an unreasonable effort: finding out what to do took longer than doing it. Most importantly, we have adhered to the heap oriented WAM approach.

8.3 The event related benchmarks

We measured the performance of the following two actions:

[4] The arguments in h1/2 are also switched: this saves WAM argument register traffic.

- the *setup* of agents
- the *execution* of agents

This terminology might differ from what is used in the B-Prolog documentation, so we clarify what we mean: for an action rule `p(Z), {event(X,M)} => Body.` the goal `p(Z)` sets up the agent. In B-Prolog terminology this means generating the agent and suspending it. The Body is not executed at this point, but in principle the matching of the head and the guard checking are included in the setup. In our benchmarks, those are absent.

A subsequent goal to `post_event(X,...)` executes all agents waiting for an event on X. This involves setting up the internal data structures to make sure the agents are called in the correct order, installing the corresponding goals, entering the rule (or clause) for the agent, and executing the body. In our benchmarks the bodies of the agent rules are empty.

The benchmarks use the following action rule:

```
true(X), {event(X,_)} => true.
```

In order to measure the setup of N agents, we execute

```
?- time(do_n_times(N,true(X))).
```

In order to measure the execution of N agents, we first set up N trivial succeeding action rules as above. Subsequently, we measure the time for performing one event post. More concretely:

```
?- do_n_times(N,true(X)), time(post_event(X,_)).
```

The cost of setup has in principle a component that is linear in the number of arguments of the agent in both approaches. The cost of the execution has a component linear in the number of arguments only in the WAM approach. It is therefore worthwhile to make the measurement for agents with different arity, i.e. for rules of the form

```
true(X,A1,A2,A3,...,An), {event(X,_)} => true.
```

for different n. The earlier mentioned true/1 rule corresponds to $n = 0$. The timings were made with B-Prolog 6.9-b4.2. The results in Table 10 are for $N = 1000$ and repetition factor 10.000.

Table 10 indicates the following:

- setup depends on the arity in both systems; hProlog is substantially faster and it seems as if this will be true for all arities
- exec is nearly independent of n in B-Prolog: the small increase might be a caching effect, since the size of the suspension frames grows of course
- in hProlog, exec clearly depends on n; however, it takes more than 20 agent arguments before B-Prolog catches up with hProlog

A (cautious) conclusion is that unless the arity of the agents is higher than twenty, the TOAM stack based implementation doesn't offer a better performance than the heap based implementation.

action	n	B-Prolog	hProlog
setup	0	938	220
setup	10	2719	2130
setup	20	4472	2715
setup	25	4952	3098
exec	0	749	341
exec	10	772	550
exec	20	777	716
exec	25	786	823

Table 10. Benchmarking the different phases of an event

9 Conclusion and Future Work

The implementation of delay in B-Prolog fits in very well with the TOAM principle of passing arguments on the execution stack. It feels as if the TOAM ought to have an advantage over the WAM for implementing delay. However, our experiments show that it is not systematically nor significantly better than a well tuned WAM implementation of delay: very important to the TOAM performance on programs using delay are specialization (in the form of abstract machine instructions) and inlining. These techniques can be applied in a WAM context, and together with some low level support (a couple of built-ins) and a dedicated slot for freeze (or for other events) which also B-Prolog uses, we get similar performance in WAM and TOAM. This paper also shows that it is worthwhile to redo and rethink old experiments. In particular, the explanation of performance differences must remain open for revision.

The battle between the TOAM and the WAM cannot be decided by performance considerations alone: on the tested benchmarks the speed difference is too small. An important advantage of the WAM approach to delay remains however: since it treats suspended goals as first class terms, it lends itself easily to experimenting with more flexible scheduling strategies for the wakeup of agents. From the implementation point of view, the WAM approach also seems more attractive, mainly because no control stack garbage collector is needed.

Our original motivation was related to compiling CHR to Action Rules. This compiler is now operational and could be used as a good source for more realistic benchmarks using events in particular. We will hopefully find the time to follow that road. Even more interesting is to incorporate *suspension frames on the heap* as sketched in Section 2 into a WAM implementation. We are currently investigating this [7].

Acknowledgements

Part of this work was conducted while the first author was a guest at the Institut de Mathématiques Appliquées of the Université Catholique de l'Ouest in Angers,

France. Sincere thanks for this hospitality. We also thank Henk Vandecasteele for maintaining the hipP compiler which we use within hProlog.

References

1. H. Aït-Kaci. The WAM: a (real) tutorial. Technical Report 5, DEC Paris Research Report, 1990.
2. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *Logic Programming: Proc. of the Fourth International Conference (Volume 1)*, pages 40–58. MIT Press, Cambridge, MA, 1987.
3. M. Carlsson. *Design and Implementation of an Or-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology (KTH), Stokholm, Sweden, Mar. 1990 See also: http://www.sics.se/isl/sicstus.html.
4. W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, 1984.
5. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Dept. of Computer Science, K.U.Leuven, Belgium, Oct. 2002.
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. Suspension frames on the heap. Report CW In Preparation, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Aug. 2006.
8. C. Holzbaur. Meta-structures vs. Attributed Variables in the Context of Extensible Unification. In M. Bruynooghe and M. Wising, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 260–268. Springer-Verlag, Aug. 1992.
9. C. Holzbaur and T. Fruhwirth. Compiling constraint handling rules into Prolog with attributed variables. In *International Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 117–133. LNCS 1702, 1999.
10. U. Neumerkel. Extensible unification by metastructures. In *Proceedings of the second workshop on Metaprogramming in Logic (META'90)*, pages 352–364, Apr. 1990.
11. T. Schrijvers, N.-F. Zhou, and B. Demoen. Translating Constraint Handling Rules into Action Rules. Third Workshop on Constraint Handling Rules, Report CW 452, K.U.Leuven, Department of Computer Science, July 2006.
12. R. Tronçon, G. Janssens, and B. Demoen. Alternatives for compile & run in the WAM. In *Proceedings of CICLOPS 2003: Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 45–58. University of Porto, 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto, December 2003.
13. D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.
14. N.-F. Zhou. On the Scheme of Passing Arguments in Stack Frames for Prolog. In *Proceedings of The International Conference on Logic Programming*, pages 159–174. MIT Press, 1994.

15. N.-F. Zhou. A Novel Implementation Method for Delay. In *Joint Internatinal Conference and Symposium on Logic Programming*, pages 97–111. MIT Press, 1996.

On Applying Deductive Databases to Inductive Logic Programming: a Performance Study

Tiago Soares, Michel Ferreira, Ricardo Rocha, and Nuno A. Fonseca

DCC-FC & LIACC
University of Porto, Portugal
{tiagosoares,michel,ricroc,nf}@ncc.up.pt

Abstract. Inductive Logic Programming (ILP) tries to derive an intensional representation of data (a *theory*) from its extensional one, which includes positive and negative examples, as well as facts from a *background knowledge*. This data is primarily available from relational databases, and has to be converted to Prolog facts in order to be used by most ILP systems. Furthermore, the operations involved in ILP execution are also very database oriented, including selections, joins and aggregations. We thus argue that the Prolog implementation of ILP systems can profit from a hybrid execution between a logic system and a relational database system, that can be obtained by using a coupled deductive database system. This hybrid execution is completely transparent for the Prolog programmer, with the deductive database system abstracting all the Prolog to relational algebra translation. In this paper we propose several approaches of coding ILP algorithms using deductive databases technology, with different distributions of work between the logic system and the database system. We perform an in-depth evaluation of the different approaches on a set of real-size problems. For large problems we are able to obtain speedups of more than a factor of 100. The size of the problems that can be solved is also significantly improved thanks to a non-memory storage of data-sets.

1 Introduction

The amount of data collected and stored in databases is growing considerably in almost all areas of human activity. A paramount example is the explosion of bio-tech data that, as a result of automation in biochemistry, doubles its size every three to six months [1]. Most of this data is structured and stored in relational databases and, in more complex applications, it can involve several relations, thus being spread over multiple tables. However, many important data mining techniques look for patterns in a single relation (or table) where each tuple (or row) is one object of interest. Great care and effort has to be made in order to squeeze as much relevant data as possible into a single table so that propositional data mining algorithms can be applied. Notwithstanding this

preparation step, propositionalizing data from multiple tables into a single one may lead to redundancy, loss of information [2] or to tables of prohibitive size [3].

On the other hand, Multi-Relational Data Mining (MRDM) systems are able to analyse data from multiple relations without propositionalizing data into a single table first. Most of the multi-relational data mining techniques have been developed within the area of Inductive Logic Programming (ILP) [4]. However, on complex or sizable applications, ILP systems suffer from significant limitations that reduce their applicability in many data mining tasks. First, ILP systems are computationally expensive - evaluating individual rules may take considerable time, and thus, to compute a model, an ILP system can take several hours or even days. Second, most ILP systems execute in main memory, therefore limiting their ability to process large databases. Efficiency and scalability are thus two of the major challenges that current ILP systems must overcome.

The main contribution of this paper is thus the proposal of applying Deductive Databases (DDB) to ILP, allowing the interface with a relational database system to become transparent to the ILP system. In particular, we will use the MYDDAS system [5], which couples YapTab [6] with MySQL [7], as the DDB system, and April [8], as the ILP system. Being able to abstract the Prolog to SQL translation, we concentrate in describing and evaluating several high-level coupling approaches, with different distributions of work between the logic system and the database system. These alternative coupling approaches correspond to different formulations in Prolog of relational operations, such as joins and aggregations, that are transparently implemented by the DDB system. We evaluate the different approaches on a set of real-size problems, showing that significant improvements in performance can be achieved by coupling ILP with DDB, and that the size of the problems solved can be significantly increased due to a non-memory storage of the data-sets.

The remainder of the paper is organised as follows. First, we discuss the main aspects of a typical coupling interface between a logic programming system and a relational database. Then, we introduce some background concepts about ILP and describe a particular ILP algorithm. Next, we show how to improve ILP algorithms efficiency by performing the coverage computation of rules with the database system. We then present some experimental results and end by outlining some conclusions.

2 Coupling a Logic System with a Relational Database

On a coupled DDB system, the predicates defined extensionally in database relations usually require a directive such as:

```
:- db_import(edge_db,edge,my_conn).
```

This directive associates a predicate `edge/2` with the relation `edge_db` that is accessible through a connection with the database system named `my_conn`. In MYDDAS, what this directive does is asserting a clause such as the one in Fig. 1.

```
edge(A,B) :-
   translate(proj_term(A,B),edge(A,B),SQLQuery),
   db_query(my_conn,SQLQuery,ResultSet),
   db_row(ResultSet,[A,B]).
```

Fig. 1. Asserted clause for an imported database predicate

Of the three predicates in Fig. 1, `translate/3`, `db_query/3` and `db_row/2`, the simplest one is `db_query/3`. This predicate uses the connection identifier, `my_conn`, to send an SQL query, `SQLQuery`, to the database system that executes the query and returns a pointer to the set of matching tuples, `ResultSet`.

The `db_row/2` predicate is more interesting. It usually navigates through the result set tuple-at-a-time using backtracking. It unifies the current tuple in the result set with the arguments of a list or some other structure. Several optimizations can be implemented for `db_row/2` [9]. The most obvious is replacing the unification by a simple binding operation for the unbound variables, since normally the SQL query already returns only the tuples that *unify* with the list arguments. Another interesting feature of `db_row/2` is how it handles pruning through a *cut* over the result set [10].

The most interesting predicate is `translate/3`, which translates a query written in logic to an SQL expression that is understood by database systems [11]. For example, the query goal '`?- edge(10,B).`' will generate the call `translate(proj_term(10,B),edge(10,B),SQLQuery)`, exiting with `SQLQuery` bound to '`SELECT 10, A.attr2 FROM edge_db A WHERE A.attr1=10`'.

The `translate/3` predicate can still be used to implement a more complex division of work between the logic system and the database system. Suppose we write the following query goal:

```
?- edge(A,B), edge(B,A).
```

A DDB system might decide that this query is more efficiently executed if the joining of the two `edge/2` goals is performed by the database system, instead of by the logic system. The system will then generate the `translate/3` call `translate(proj_term(A,B),(edge(A,B),edge(B,A)),SQLQuery)` to obtain the query '`SELECT A.attr1, A.attr2 FROM edge_db A, edge_db B WHERE B.attr1=A.attr2 AND B.attr2=A.attr1`'. In MYDDAS, we use the `db_view/3` primitive to accomplish this. For the example given above, we should use a directive like `db_view(direct_cycle(A,B),(edge(A,B),edge(B,A)),my_conn)`, which will assert a clause for `direct_cycle/2` in a similar way to what was done for the `db_import/3` directive in Fig. 1. This is known as *view-level access*, in opposition to the previous *relation-level access*.

The `translate/3` predicate also allows specifying database logic goals that include higher-order operations, such as aggregate functions that compute values over sets of attributes. Although higher-order operations are not part of the relational database model, virtually every database system supports aggregate functions over relations, such as `sum()`, `avg()`, `count()`, `min()` and `max()` which

compute the sum, the average, the number, the minimum and the maximum of given attributes.

In `translate/3`, aggregate functions are represented as a binary subgoal in the database goal, mapping the predicate symbol of such subgoal to the aggregate function name in the database system. The first argument of the subgoal is mapped to the attribute over which the aggregate function is to be computed and the second argument specifies the relation goal. The projection term is specified to include the result of the aggregation. As an example, if we want `translate/3` to generate an SQL query to compute the number of tuples from predicate `edge/2` that depart from point `10` we would write:

```
?- translate(count(X),(X is count(B,edge(10,B))),SQLQuery).
```

and this would bind `SQLQuery` to `'SELECT COUNT(A.attr2) FROM edge_db A WHERE A.attr1=10'`.

3 Inductive Logic Programming

The normal problem that an ILP system must solve is to find a consistent and complete *theory*, from a set of examples and prior knowledge, the *background knowledge*, that explains all given positive examples, while being consistent with the given negative examples [4]. In general, the background knowledge and the set of examples can be arbitrary logic programs. We next describe ILP execution in more detail by using the classical *Michalski train problem* [12].

In the Michalski train problem the theory to be found should explain why trains are travelling eastbound. There are five examples of trains known to be travelling eastbound, which constitutes the set of positive examples, and five examples of trains known to be travelling westbound, which constitutes the set of negative examples. All our observations about these trains, such as size, number, position, contents of carriages, etc, constitutes our background knowledge. We present in Fig. 2 the set of positive and negative examples, together with part of the background knowledge (describing the train *east*1).

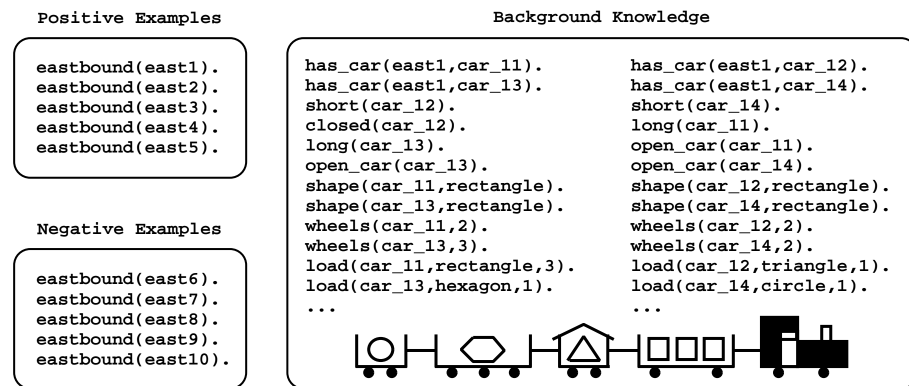


Fig. 2. Examples and background knowledge for the Michalski train problem

To derive a theory with the desired properties, many ILP systems follow some kind of *generate-and-test* approach to traverse the *hypotheses space* [13,14]. A general ILP system spends most of its time evaluating hypotheses, either because the number of examples is large or because testing each example is computationally hard. For instance, a possible sequence of hypotheses (clauses) generated for the Michalski train problem would be:

```
eastbound(A) :- has_car(A,B).
eastbound(A) :- has_car(A,C).
eastbound(A) :- has_car(A,D).
eastbound(A) :- has_car(A,E).
eastbound(A) :- has_car(A,B), short(B).
eastbound(A) :- has_car(A,B), open_car(B).
eastbound(A) :- has_car(A,B), shape(B,rectangle).
eastbound(A) :- has_car(A,B), wheels(B,2).
eastbound(A) :- has_car(A,B), load(B,circle,1).
...
```

For each of these clauses the ILP algorithm computes its *coverage*, that is, the number of positive and negatives examples that can be deduced from it. If a clause covers all of the positive examples and none of the negative examples, then the ILP system stops. Otherwise, an alternative stop criteria should be used, such as, the number of clauses evaluated, the number of positive examples covered, or time. A simplified algorithm for the coverage computation of a clause is presented next in Fig. 3. In the evaluation section we name this approach the *Basic ILP Approach*.

Consider now that we call the `compute_coverage/3` predicate for clause '`eastbound(A) :- has_car(A,B), short(B).`'. Initially, it starts by asserting the clause to the program code, resetting a counter `pos`, and by calling the predicate representing the positive examples. The `positive_examples/1` predicate binds variable `X` to the first positive example, say `east1`, and the `process/3` predicate creates the `eastbound(east1)` goal, which is called using the `once/1` primitive. The `once/1` primitive is used to avoid backtracking on alternative ways to derive the current goal. It is defined in Prolog as '`once(Goal) :- call(Goal), !.`'. If the positive example succeeds, counter `pos` is incremented and we force failure. Failure, whether forced or unforced, will backtrack to alternative positive examples, traversing all of them and counting those that succeed. The process is repeated for negative examples and finally the asserted clause is retracted.

4 Coupling ILP with a Deductive Database System

The time spent in the coverage computation of the rules generated by an ILP system represents the larger percentage of the total execution time of such systems. In this section we describe several approaches to divide the work between the

```
compute_coverage(Clause,ScorePos,ScoreNeg) :-
   assert(Clause),
   reset_counter(pos,0),
   (
      positive_examples(X),
      process(Clause,X,GoalP),
      once(GoalP),
      incr_counter(pos),
      fail
   ;
      true
   ),
   counter(pos,ScorePos),
   reset_counter(neg,0),
   (
      negative_examples(Y),
      process(Clause,Y,GoalN),
      once(GoalN),
      incr_counter(neg),
      fail
   ;
      true
   ),
   counter(neg,ScoreNeg),
   retract(Clause).
```

Fig. 3. Coverage computation

logic system and the relational database system in order to maximize overall efficiency of the coverage computation. We will present this coverage computation starting with its original implementation, and then incrementally transferring computational work from the logic system to the database system. In what follows, we name each of the approaches in order to compare their performance in the evaluation section.

4.1 Relation-Level Approach

In coupled DDB systems the level of transparency allows the user to deal with relationally defined predicates exactly as if they were defined in the Prolog source code. Basically, predicates are transparently mapped to database tables or views. The extensionally defined predicates are mapped to tables, while the intensionally defined predicates are mapped to views. This mapping scheme provides a transparent solution for the designer of the ILP engine (if the system is implemented in a first order language like Prolog). However, it results in increased communication with the relational database system since many accesses are made to evaluate each single clause.

In particular, the `compute_coverage/3` predicate of Fig. 3 can be used when the background knowledge and the positive and negative examples are declared

through the `db_import/3` directive. However, using only the `db_import/3` directives, the coverage computation uses *relation-level access* to retrieve the tuples from the database system. This means an uneven division of work between the logic system and the database system. We name this approach the *Relation-Level Approach*.

4.2 View-Level Approach

A fundamental improvement to the *Relation-Level Approach* is to transfer the joining effort of the background knowledge goals, in the body of the current clause, to the database system.

In MYDDAS, we use the `db_view/3` predicate to convert the relation-level accesses in *view-level accesses*, as explained in section 2. Following our previous example, instead of asserting the clause '`eastbound(A) :- has_car(A,B), short(B).`', we now create the view using the directive `db_view(view(A,B), (has_car(A,B),short(B)),my_conn)` and assert the clause '`eastbound(A) :- view(A,B).`'. As the view just has to outlive the coverage computation of the current clause, an useful optimization is to use a predicate `run_view/3` which calls the view without asserting it: '`eastbound(A) :- run_view(view(A,B), (has_car(A,B),short(B)),my_conn).`'. The coverage computation algorithm of Fig. 3 works as before, with the joining computation performed now by the database system. We name this approach the *View-Level Approach*.

4.3 View-Level/Once Approach

Some very important issues in using the database system to compute the join of the goals in the body of the current clause and the algorithm of Fig. 3 arise for the `once/1` primitive. Not only the coupling interface must support deallocation of queries result sets through a '`!`' [10], but also the pruning of unnecessary computation work to derive alternative solutions is not being done by `once/1`, as intended. The database system has already computed all the alternative solutions when the '`!`' is executed. Reducing the scope of the join is thus fundamental and, for a given positive or negative example, the database system only needs to compute the first tuple of the join.

In order to reduce the scope of the join computed by the database system, we should push the `once/1` call to the database view. Therefore, the asserted clause includes a `once/1` predicate on the view definition which we can efficiently translate to SQL using the '`LIMIT 1`' SQL keyword. For our example, the asserted clause is now: '`eastbound(A) :- run_view(view(A,B),once(has_car(A,B), short(B)),my_conn)`'. For the first positive example `east1` the SQL expression generated for the view is: '`SELECT A.attr1, A.attr2 FROM has_car_db A, short_db B WHERE A.attr1='east1' AND B.attr1=A.attr2 LIMIT 1`'.

We can now drop the `once/1` primitive from the call on the code of Fig. 3. We name this approach the *View-Level/Once Approach*.

4.4 Aggregation/View Approach

A final transfer of computation work from the logic system to the database system can be done for the aggregation operation which counts the number of examples covered by a rule. The *Basic ILP Approach* uses extra-logical global variables to perform this counting operation, as it would be too inefficient without this feature.

To transfer the aggregation work to the database system we need to restrict the theories we are inducing to non-recursive theories, where the head of the clause can not appear as a goal in the body. With this restriction, we can drop the assertion of the current clause to the program code and use the `db_view/3` predicate with the aggregation operation `count/1` on an attribute of the relation holding the positive or negative examples. Also, the view now includes the positive or negative examples relation as a goal co-joined with the goals in the body of the current clause. Again, the join should only test for the existence of one tuple in the body goals for each of the examples. We introduce a predicate `exists/1`, similar to `once/1`, extending again the Prolog to SQL compiler, which will be translated to an SQL expression involving an existential sub-query. For our example clause, '`eastbound(A) :- has_car(A,B), short(B).`', the view used to compute positive coverage would be the following:

```
db_view(count_examples(P),
        P is count(A,(eastbound(A),exists(has_car(A,B),short(B)))),
        my_conn).
```

which generates the SQL expression:

```
SELECT COUNT(A.attr1) FROM eastbound_db A
WHERE EXISTS (SELECT * FROM has_car_db B, short_db C
              WHERE B.attr1=A.attr1 AND B.attr2=C.attr1 LIMIT 1)
```

Although the '`LIMIT 1`' primitive may seem redundant for an existential sub-query, our experiments showed that MySQL performance is greatly improved if we include it on the sub-query. We name this approach the *Aggregation/View Approach*. Figure 4 presents a simplified algorithm for the coverage computation using this approach.

```
compute_coverage(':-'(Head,Body), ScorePos, ScoreNeg, Conn) :-
   process(pos, Head, HeadPos, AggrArgPos),
   run_view(count_positive_examples(ScorePos),
          (ScorePos is count(AggrArgPos,(HeadPos,exists(Body)))), Conn),
   process(neg, Head, HeadNeg, AggrArgNeg),
   run_view(count_negative_examples(ScoreNeg),
          (ScoreNeg is count(AggrArgNeg,(HeadNeg,exists(Body)))), Conn).
```

Fig. 4. Coverage computation with the database

Although our coverage computation predicate is very simple to implement in the context of a DDB, it brings with it a complex set of features which have

been the subject of recent research in the implementation of ILP systems. The first of these features is efficient higher-order computation. Reasoning about a set of values is typically inefficient in Prolog, as we usually have to build the set of values and then traverse them again to compute the desired function. This can be overcome, as shown in Fig. 3, using non-logical extensions such as global variables. Database systems have efficient aggregation algorithms.

A second feature is powerful indexing. Typical Prolog systems indexing is restricted to the first argument of clauses. The inefficiency of this indexing scheme for ILP algorithms, where efficient selections and joins are fundamental, motivated the development of the just-in-time, full arguments, indexing of the Yap Prolog system 5.0 [15]. Database systems allow the creation of a variety of index types over all the attributes of a relation.

A third feature is goal-reordering. The coverage computation goal is just to compute the number of positive and negative examples covered by a clause. The execution order of the goals in the body of a clause is irrelevant. Query optimization of database systems executes the computation of the join on the involved relations in the most efficient way, using transformations which are similar to goal-reordering in Prolog execution.

Another feature is parallelism. By using a parallel database system we can have the aggregation, selection and joining operations implemented using the parallel algorithms of parallel database systems.

5 Performance Evaluation

In order to evaluate and analyse the different approaches for coverage computation, we used the April ILP system [8] to obtain sets of hypotheses which are generated during the ILP algorithm search process. We then implemented the five different approaches for coverage computation through simple Prolog programs, as explained in the previous sections, and measured the time taken by each on the different sets of hypotheses. Implementing the different approaches in April and using April's execution time as the measure, did not allow us to do a precise performance evaluation. April can use different heuristics, optimizations and search strategies, and the time taken in the search process can mislead the real speed-up obtained in the different coverage computation approaches described in this paper.

We used MYDDAS 0.9, coupling Yap 5.1.0 with MySQL Server 4.1.5-gamma, on a AMD Athlon 64 Processor 2800+ with 512 Kbytes cache and 1 Gbyte of RAM. Yap performs a just-in-time, full arguments, indexing. We have used five ILP problems: the Michalski train problem [12] and four artificially generated problems [16]. Table 1 characterizes the problems in terms of number of examples, number of relations in the background knowledge, number of clauses generated, and number of tuples.

The clauses were randomly generated and equally distributed by length, ranging from 1 to the maximum number of relations. The clauses were then evaluated using each of the described approaches.

Problem	Examples	Relations	Clauses	Tuples
train	10	10	68	240
p.m8.l27	200	8	495	321,576
p.m11.l15	200	11	582	440,000
p.m15.l29	200	15	687	603,000
p.m21.l18	200	21	672	844,200

Table 1. Problems characterization

5.1 Coverage Performance

Table 2 shows the best execution time of five runs, in seconds, for coverage computation using our five approaches in each ILP problem.

Approach	Problem				
	train	**p.m8.l27**	**p.m11.l15**	**p.m15.l29**	**p.m21.l18**
Basic ILP	0.002	15.331	50.447	33,972.225	>1 day
Relation-Level	0.515	35,583.984	>1 day	>1 day	>1 day
View-Level	0.235	n.a	n.a	n.a	n.a
View-Level/Once	0.208	99.837	628.409	2,975.051	33,229.210
Aggregation/View	0.105	5.330	14.850	251.192	734.800

Table 2. Coverage performance for the different approaches

The **train** problem is a toy problem, useful to explain how an ILP algorithm works, but totally non-typical with respect to actual problems approached through ILP. The background knowledge together with the positive and negative examples totals less than 300 tuples (facts). This number clearly fits in memory and the communication overhead with a database represents most of the execution time, as the select or join operations involve very few tuples. We included this example as it is the only one where we could obtain execution times for all of the approaches. With regard to the database approaches, this example already shows gradual improvements as the computation work is transferred to the database engine, from 0.515 seconds using the *Relation-Level Approach* to 0.105 seconds using the *Aggregation/View Approach*. For this example, all the queries are executed almost instantly, and the time difference just translates the number of queries that are sent to the database system, which decreases from the *Relation-Level Approach* to the *Aggregation/View Approach*. Even sending a no-action query to the database system and obtaining the result set, takes a core execution time, which explains the difference to the *Basic ILP Approach* for this very small problem.

For the larger problems, the core time of communication between the logic system and the database system becomes diluted as we increase the computation work of the database system. For problems involving thousands of tuples in a number of relations, the *Relation-Level Approach* is unrealistic. This approach does not transfer any computation work to the database system, other than selecting tuples from individual relations. Also, the number of queries generated is a factor of the number of tuples in each relation, which explains execution times of days or weeks for problems larger than **p.m8.l27**.

For the *View-Level Approach*, as expected, we could not obtain the execution times for the large problems, due to insufficient memory to compute the joins involved. Note that this approach does not implement the `once/1` optimization, therefore the entire join is computed instead of just the first tuple. MySQL runs out of memory when trying to compute a join of several relations, each with thousands of tuples.

For the *View-Level/Once Approach* the scope of the join is now reduced to compute just the first tuple. For problem **p.m11.l15** the slow-down factor compared to the *Basic ILP Approach* is explained by the number of queries that are sent to the database system, one for every positive and negative example. For the **p.m*** problems this means that for each of the 688 clauses a total of 200 queries (the number of positive and negative examples) are sent to the database system. As the size of the joins grows larger, as with problem **p.m15.l29**, the core time of these 200 queries becomes irrelevant compared to the time taken for computing the joins. This and the huge amount of backtracking performed by the *Basic ILP Approach* for the two largest problems, explains the speedup obtained with this approach.

On the *Aggregation/View Approach* only two queries are sent to the database system per clause, one to compute positive coverage and one to compute negative coverage. All the coverage computation work is transferred to the database system, and the core time of sending and storing the result set for just two queries is insignificant. The performance gains over the *Basic ILP Approach* are clear: a 2.8 speedup for problem **p.m8.l27**, a 3.4 speedup for problem **p.m11.l15** and a 135 speedup for the **p.m15.l29**. These results show a clear tendency for higher speedups as the size of the problems grow.

The results obtained with the *Aggregation/View Approach* are very significant. Not only we can now handle problems of size two orders of magnitude larger, thanks to the non-memory storage of data-sets, but we are also able to improve the coverage computation execution time by a very significant factor.

5.2 Coverage Analysis

The results presented in the previous section compare the different approaches for the coverage computation. In order to achieve a deeper insight on the behaviour of the DDB system usage, and therefore clarify some of the results obtained, we present in Table 3 data related to several activities of the coverage computation. These statistics were obtained by introducing a set of counters to measure the several activities. The columns in this table have the following meaning:

transl: the percentage of time spent on the `translate/3` predicate translating Prolog to SQL.

server: the percentage of time spent by the database server processing queries.

transf: the percentage of time spent in transferring result sets from the database to Prolog.

db_row: the percentage of time spent on the `db_row/2` predicate. It measures the time spent in unifying the results of the queries with the variables of the logic system.

prolog: the percentage of time spent on normal Prolog execution and not measured by the other activities.

queries: the total number of queries sent to the database server by the Prolog process.

rows: the total number of rows returned for the queries made. In parenthesis, it shows the amount of data transferred in KBytes.

Problem/Approach	**Activities**						
	transl	**server**	**transf**	**db_row**	**prolog**	**queries**	**rows**
train							
Relation-Level	9.4%	67.6%	2.0%	1.9%	19.1%	3402	5402(53)
View-Level/Once	9.6%	69.3%	1.3%	1.4%	18.4%	924	1362(7)
Aggregation/View	47.0%	41.4%	0.6%	0.5%	10.6%	154	154(0)
p.m08.l27							
View-Level/Once	4.6%	89.5%	0.3%	0.2%	5.5%	100378	236800(998)
Aggregation/View	1.5%	97.3%	0.1%	0.0%	1.1%	990	990(2)
p.m11.l15							
View-Level/Once	1.2%	97.2%	0.1%	0.1%	1.4%	117776	254000(1090)
Aggregation/View	0.8%	98.6%	0.0%	0.0%	0.6%	1164	1164(3)
p.m21.l18							
View-Level/Once	0.0%	99.9%	0.0%	0.0%	0.1%	100378	264055(1155)
Aggregation/View	0.0%	99.9%	0.0%	0.0%	0.0%	1344	1344(3)

Table 3. Activities analysis for the different approaches

For the **train** problem, the time spent on the database server is comparatively small to the other data-sets. The queries calculated are very small and easy to compute, so the other activities of the coverage computation gain relevance.

Considering only the problems that have a more interesting size, the data-set **p.m08.l27** presents the highest percentage of time spent on the **transl** activity. This can be explained by the fact that **p.m08.l27** is the smallest problem. As the size of the problems grow, the total execution time increases, therefore lowering the impact of the `translate/3` predicate on the final execution time. In fact, test results show that for each different type of approach, the core time spent on this predicate, is of the same order for any of the problems experimented. For the *Aggregation/View Approach* the times obtained were around 100 milliseconds, having a variance increase due to an enlargement of the logic clauses to be translated, as the data-sets grow in size. Notice also that in this approach only 2 queries are made to the database server per clause, one to count the positive examples that are covered, and another for the negative ones. On the *View-Level/Once Approach*, the number of queries sent to the server for each clause is augmented to 1 query per positive and negative example, which causes the time spent on the **transl** activity to increase.

Table 3 shows that the most significant part of the time is consumed in the **server** activity. Improving the efficiency of the database server in the execution

of queries is thus fundamental to achieve good results. As ILP problems often use some kind of mode declarations to supply information concerning the arguments of each predicate that may appear in the background knowledge, we use the mode information to automatically create indexes in the database in order to optimize query execution. Without indexing, the final execution time can increase more than 5000 times.

Table 3 also shows that for these approaches, only a small percentage of time is spent transferring the result sets from the database server to the Prolog system (**transf** activity). This is due to the small number of rows returned, and consequently, small amount of data transferred. This also shows that it is in fact the relational system that processes most of the work of the coverage computation. However, for the *View-Level/Once Approach*, the database returns more results than for the *Aggregation/View Approach*, which increases the time spent on unifying logic variables increases (**prolog** activity).

With respect to the **db_row** activity, we can see that for the *View-Level/Once Approach* and *Aggregation/View Approach* the time spent in this activity is not relevant. Results obtained for these approaches show that, on average, the time spent on the `db_row/2` predicate unifying the results of the queries with the logic variables, is around 2 and 400 milliseconds respectively for the *Aggregation/View Approach* and *View-Level/Once Approach*. Remember that the *Aggregation/View Approach* only returns two values for each query, while the *View-Level/Once Approach* produces far more results.

6 Conclusions and Future Work

In this work we have proposed to couple ILP systems with DDB systems. This strategy allows bringing to ILP systems the technology of relational database systems, which are very efficient in dealing with large amounts of data. Coupling ILP with DDB allows abstracting the Prolog to SQL translation from the ILP system. The ILP system just uses high-level Prolog predicates that implement relational operations that are more efficiently executed by the relational database system. We argue that this strategy is easier to implement and maintain than the approach that tries to incorporate database technology directly in the logic programming system. And, much more important, it allows a substantial increase of the size of the problems that can be solved using ILP since the data does not need to be loaded to memory by the ILP systems.

The performance results in execution speed for coverage computation are very significant and show a tendency to improve as the size of the problems grows. The size of the problems is exactly our most significant result, as the storage of data-sets in database relations allows an increase of more than 2 orders of magnitude in the size of the problems than can be approached by ILP systems.

In the future we plan to deal with recursive theories, through the YapTab tabling system [6], and to be able to send packs of clauses as a single query to the database system, using its grouping operators, to avoid redundant computation. We also plan to use the query packs technique [17], which is very similar to

the tabling of prefixes technique [18], to perform a many-at-once optimization of SQL queries. Not only do some database systems perform caching of queries, but it is also simple to implement similar techniques to query-packs on a DDB context.

A more ambitious future goal aims at a full integration of April and MYDDAS in a single programming environment where any program can be seen as a set of extensional data represented in a database, a set of intensional (and extensional) data represented by logic rules, and a set of undefined data that the ILP component of the system should be able to derive and compile to intensional data to be used by the program itself.

Acknowledgements

This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia (FCT) and Programa POSC. Tiago Soares is funded by FCT PhD grant SFRH/BD/23906/2005. Michel Ferreira was funded by FCT sabbatical grant SFRH/BSAB/518/2005, and thanks Manuel Hermenegildo and University of New Mexico for hosting his research.

References

1. Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T.N., Weissig, H., Shindyalov, I.N., Bourne, P.E.: The protein data bank. Nucleic Acids Research (2000) 235–242
2. Wrobel, S.: Inductive Logic Programming for Knowledge Discovery in Databases. In: Relational Data Mining. Springer-Verlag (2001) 74–101
3. Raedt, L.D.: Attribute Value Learning versus Inductive Logic Programming: The Missing Links. In: Inductive Logic Programming. Volume 1446 of LNAI., Springer-Verlag (1998) 1–8
4. Muggleton, S., Raedt, L.D.: Inductive Logic Programming: Theory and Methods. Journal of Logic Programming **19/20** (1994) 629–679
5. Soares, T., Ferreira, M., Rocha, R.: The MYDDAS Programmer's Manual. Technical Report DCC-2005-10, Department of Computer Science, University of Porto (2005)
6. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
7. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
8. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)
9. Ferreira, M., Rocha, R., Silva, S.: Comparing Alternative Approaches for Coupling Logic Programming with Relational Databases. In: Colloquium on Implementation of Constraint and LOgic Programming Systems. (2004) 71–82

10. Soares, T., Rocha, R., Ferreira, M.: Generic Cut Actions for External Prolog Predicates. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 16–30
11. Draxler, C.: Accessing Relational and Higher Databases Through Database Set Predicates. PhD thesis, Zurich University (1991)
12. Michalski, R.S., Larson, J.B.: Selection of Most Representative Training Examples and Incremental Generation of VL918 Hypotheses: The Underlying Mehtodology and the Description of Programs ESEL and AQ11. Technical Report 867, Department of Computer Science, University of Illinois at Urbana-Champaign (1978)
13. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A Midterm Report. In: European Conference on Machine Learning. Volume 667., Springer-Verlag (1993) 3–20
14. Muggleton, S., Firth, J.: Relational Rule Induction with CProgol4.4: A Tutorial Introduction. In: Relational Data Mining. Springer-Verlag (2001) 160–188
15. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. (2006) Available from `http://www.ncc.up.pt/~vsc/Yap`.
16. Botta, M., Giordana, A., Saitta, L., Sebag, M.: Relational Learning as Search in a Critical Region. Journal of Machine Learning Research **4** (2003) 431–463
17. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16** (2002) 135–166
18. Rocha, R., Fonseca, N., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning. Number 3720 in LNAI, Springer-Verlag (2005) 707–714

DBTAB: a Relational Storage Model for the YapTab Tabling System

Pedro Costa, Ricardo Rocha, and Michel Ferreira

DCC-FC & LIACC
University of Porto, Portugal
c0370061@dcc.fc.up.pt {ricroc,michel}@ncc.up.pt

Abstract. Resolution strategies based on tabling have proved to be particularly effective in logic programs. However, when tabling is used for applications that store large answers and/or a huge number of answers, we can quickly run out of memory. In general, to recover space, we will have no choice but to delete some of the tables. In this work, we propose an alternative approach and instead of deleting tables, we store them externally using a relational database system. Subsequent calls to stored tables would import answers from the database, hence avoiding recomputation. To validate our approach, we have extended the YapTab tabling system to provide engine support for exporting and importing tables to and from the MySQL relational database management system.

1 Introduction

Tabling [1] is an implementation technique where intermediate answers for subgoals are stored and then reused when a repeated call appears. Resolution strategies based on tabling [2,3] have proved to be particularly effective in logic programs, reducing the search space, avoiding looping and enhancing the termination properties of Prolog models based on SLD resolution [4].

The performance of tabling largely depends on the implementation of the table itself; being called upon very often, fast look up and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [5]. Tries are trees in which there is one node for every common prefix [6]. Tries have proved to be one of the main assets of tabling implementations, because they are quite compact for most applications while having fast look up and insertion. The YapTab tabling system [7] uses tries to implement tables.

When tabling is used for applications that build many queries or that store a huge number of answers, we can build arbitrarily many and possibly very large tables, quickly filling up memory. In general, there is no choice but to throw away some of the tables (ideally, the least likely to be used next). The common control implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically abolish some of the tables.

A more recent proposal, is the approach implemented in YapTab, where a memory management strategy, based on a *least recently used* algorithm, automatically recovers space from the least recently used tables when the system

runs out of memory. With this approach, the programmer can still force the deletion of particular tables, but can also rely on the effectiveness of the memory management algorithm to completely avoid the problem of deciding what potentially useless tables should be deleted. Note that, in both situations, the loss of stored answers within the deleted tables is unavoidable, leading to the need of restarting the evaluation whenever a repeated call occurs.

In this work, we propose an alternative approach and instead of deleting tables, we store them externally using a relational database management system (RDBMS). Later, when a repeated call appears, we load the stored answers from the database, hence avoiding recomputing them. With this approach, we can still use YapTab's memory management algorithm, but to decide what tables to move to database storage when the system runs out of memory, instead of using it to decide what tables to delete.

To validate this approach we thus propose DBTAB, a relational model for representing and storing tables externally in tabled logic programs. In particular, we will use YapTab as the tabling system and MySQL [8] as the RDBMS. The initial implementation of DBTAB only handles atomic terms such as integers, atoms and floating-point numbers.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce our model and discuss how tables can be represented externally in database storage. We then describe how we extended YapTab to provide engine support for exporting and importing answers to and from the RDBMS. At the end, we present initial experimental results and outline some conclusions.

2 The Table Space

Tabled programs are evaluated by storing all found answers for current subgoals in a proper data space, the *table space*. Whenever a subgoal $\mathcal{S}$ is called for the first time, a matching entry is allocated in the table space and every generated answer for the subgoal is stored under this entry. Repeated calls to $\mathcal{S}$ or its *variants*[1] are resolved by consumption of these previously stored answers. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. When all possible resolutions are performed, $\mathcal{S}$ is said to be *completely evaluated*.

The table space can be accessed in a number of ways: **(i)** to look up if a subgoal is in the table, and if not insert it; **(ii)** to verify whether a newly found answer is already in the table, and if not insert it; and, **(iii)** to load answers to variant subgoals. Two levels of tries are used to implement tables, one for subgoal calls, other for computed answers. Each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the *subgoal trie*. Every subgoal call is represented in this trie as an unique path to a *subgoal frame* data structure, with argument terms stored within the internal nodes. Terms with

[1] Two calls are said to be variants if they are the same up to variable renaming.

common prefixes branch off each other at the first distinguishing symbol. If free variables are present within the arguments, all possible bindings are stored into the *answer trie*, i.e., all possible answers to the subgoal are mapped to unique paths in this second trie. When inserting new answers, only the substitutions for the unbound variables in the subgoal call are stored. This optimization is called *substitution factoring* [5].

Tries are implemented by representing each trie node by a data structure with four fields each. The first field (`TrNode_symbol`) stores the symbol for the node. The second (`TrNode_child`) and third (`TrNode_parent`) fields store pointers respectively to the first child node and to the parent node. The fourth field (`TrNode_next`) stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and then the list of sibling pointers.

An example for a tabled predicate $f/2$ is shown in Fig. 1. Initially, the subgoal trie only contains the root node. When the subgoal $f(X, a)$ is called, two internal nodes are inserted: one for the variable X, and a second last for the constant a. Notice that variables are represented as distinct constants, as proposed by Bachmair *et al.* [9]. The subgoal frame is inserted as a leaf, waiting for the answers to appear. Then, the subgoal $f(Y, 1)$ is inserted. It shares one common node with $f(X, a)$, but the second argument is different so a different subgoal frame needs to be created.

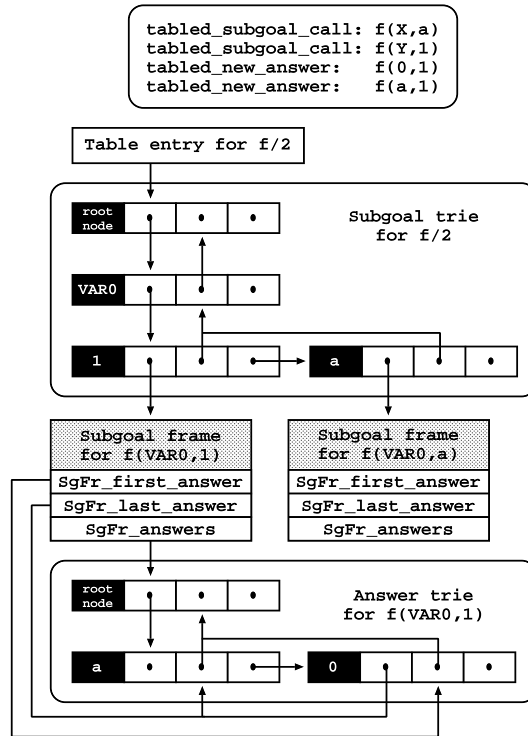


Fig. 1. Using tries to organize the table space

At last, the answers for $f(Y, 1)$ are stored in the answer trie as their values are computed. The leaf answer nodes are chained in a linked list in insertion time order (using the `TrNode_child` field), so that recovery may happen the same way. Finally, the subgoal frame internal pointers `SgFr_first_answer` and `SgFr_last_answer` are set to point respectively to the first and last answer of this list. Thus, when consuming answers, a variant subgoal needs only to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

3 The Relational Storage Model

The chosen RDBMS for DBTAB is MySQL 4.1 [8], running a InnoDB storage engine. This is a transaction-safe engine with commit, rollback and crash-recovery capabilities. InnoDB tables present no limitation in terms of growth and support FOREIGN KEY constraints with CASCADE abilities. The MySQL C API for prepared statements is used to manipulate the record-sets, benefiting of several advantages in terms of performance: the statement is parsed only once, when it is first sent; network traffic is substantially reduced since statement invocation require only the respective input parameters; a binary protocol is used to transfer data between the client and the server.

3.1 Representing the Table Space

Figure 2 shows the ER diagram for the relational representation of the table space, the DBTAB database. The diagram is divided in two types of tables: system tables, identified by the prefix DBTAB, and predicate tables, identified by the prefix SESSION. System tables basically maintain control status, while predicate tables are meant to hold look-up values and run-time data, such as computed answers and meta-information about known subgoals.

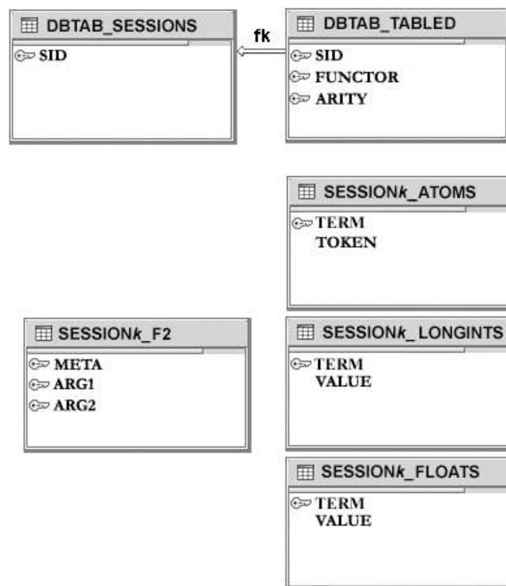


Fig. 2. The DBTAB relational schema

Storing run-time data, of possible multiple sources, raises the important issue of multi-user concurrency. Since the same database is to be used as a final repository of data, each running instance of YapTab must be uniquely identified in order to refer to its own found answers. To tackle this problem, DBTAB introduces the notion of *session*. A specific predicate, `tabling_init_session/1`, is introduced to initialize sessions. It takes one argument that is considered to be a *session id*, that can be either a free variable or a ground term. In the first case, a new identifying integer is attained from the DBTAB_SESSIONS table and unified with the variable. On the other hand, if the argument is a ground integer, its value is searched in the sessions table, and should it be found, the indicated session is reestablished[2]. The

[2] Currently, this means only that the same session id is reused.

`tabling_kill_session/0` predicate can be used to finish the opened session and clean-up all of its dependent information.

Tabling begins by the identification of the predicates to handle. YapTab's directive '`:- table p/n.`' is used for this purpose, generating a new table entry data structure for the specified predicate and inserting it into the table space. DBTAB extends the previously described behaviour by registering functor p and arity n into DBTAB_TABLED. The SID field acts as a foreign key to DBTAB_SESSIONS, establishing the dependency net for the session. DBTAB then dynamically creates a new relational table SESSIONk_PN to hold all completed answer tries for p/n, together with three auxiliary relational tables SESSIONk_ATOMS, SESSIONk_LONGINTS and SESSIONk_FLOATS, with k being the current session id. Along with all computed answers, a meta-representation for every p/n's completely evaluated subgoal is to be stored within the SESSIONk_PN table. The integer field META is used to tell apart these two kinds of records: a zero value signals an answer; a positive non-zero value signals a meta-information record. The arguments of p/n are mapped into integer fields[3] named ARGi, with i being an index between 1 and n.

The `abolish_table(p/n)` predicate is used by YapTab to remove p/n's entry from the table space. DBTAB's expansion of this predicate deletes the corresponding record from DBTAB_TABLED and drops all the session tables associated with p/n. Both in YapTab and DBTAB, the `abolish_all_tables/0` predicate can be used to dispose of all table entries: the action takes place as if `abolish_table/1` was called for every tabled predicate.

3.2 Handling Primitive Types

YapTab handles atomic terms such as integers, atoms and floating-point numbers. YapTab determines the type of each term by reading its mask bits. The non-mask part of a term is thus less than the usual 32 or 64-bit representation, so an additional term is used to represent floating-point values and integers greater than the maximum masked integer allowed. In what follows we call these integers *long integer terms*. Due to this difference in sizes, internal representation at trie level may require more than one node. While integer and atom terms use only one node, long integer and floating-point terms use 3 and 4 nodes.

DBTAB explores this idea and handles answer terms dividing them in two categories: *atomic terms* have their values directly stored within the corresponding ARGi record fields; *non-atomic terms* are substituted in the ARGi record fields by unique sequential values that work as a foreign key to the TERM field of the auxiliary tables for the predicate. These sequential values are masked as YapTab terms in order to simplify the loading algorithm[4]. Atomic terms com-

[3] The YapTab internal representation of terms can be thought of as 32 or 64-bit long integers, so MySQL INTEGER or BIGINT types are accordingly used to store these values.

[4] The `YAP_MkApplTerm()` macro is used to create *dummy* application terms and then the non-tag part of these terms is used to hold the sequential value.

prise integers and atoms, while floating-point and long integer terms are considered non-atomic. Two auxiliary tables are defined to hold non-atomic values: SESSIONk_FLOATS stores floating-point values used to build floating-point terms; SESSIONk_LONGINTS is used to store long integer values.

Aside from storing atoms into the predicate tables, every session stores their YapTab's internal representation as well as their string values, respectively in the TERM and TOKEN fields of its SESSIONk_ATOMS table. When reestablishing a session, this table is used to rebuild the previous internal symbol addressing space.

3.3 Manipulating Data Through Prepared Statements

Data exchange between the database and YapTab is done through the MySQL C API along with a *prepared statement* data structure (see Fig. 3 for details). The SQL statements used to store/retrieve information are sent to the database for parsing, and, on success, the returned handle is used to initialize the `statement` pointer. Additional information about possible used parameters is stored within the sub-structure `params`.

If a record-set is to be returned upon the statement's execution, the result-set pointer `records.metadata` and the sub-structure `fields` are initialized with information regarding it. For predicate tables, the `stmt_buffer` pointer will be initialized with an integer array, sized to hold an entire record. Since the `params` and `fields` sub-structures are never used at the same time, they can share the `stmt_buffer`, `bind`, `null`, and `length` arrays - these arrays are sized accordingly to the largest requirement in terms of size. After record storing, the `records.num_rows` sub-structure holds the count of retrieved rows.

```
typedef struct prepared_statement {
  MYSQL_STMT     *statement;
  my_ulonglong   affected_rows;
  void           *stmt_buffer;
  struct {
    int          count;
    MYSQL_BIND   *bind;
    my_bool      *null;
    my_ulong     *length;
  } params;
  struct {
    int          count;
    MYSQL_BIND   *bind;
    my_bool      *null;
    my_ulong     *length;
  } fields;
  struct {
    MYSQL_RES    *metadata;
    my_ulonglong num_rows;
  } records;
} *PreparedStatement;
```

Fig. 3. The prepared statement structure

The table entry data structure is augmented with a pointer to a generic INSERT prepared statement. All subgoals branches hanging from this table entry share the same prepared statement. Computed answers are stored by instantiating its input parameters as required and executing it. The subgoal frame data structure is augmented with a pointer to a specific SELECT prepared statement. Ground terms in the subgoal trie are used in the refinement of the WHERE clause; the corresponding fields are not selected for retrieval since their values are already known.

Figure 4 shows the prepared statements generated to store and recover the $f(Y,1)$ subgoal call introduced back at Fig. 1. The first statement is the generic INSERT statement that is used for all insertions into table SESSIONk_F2. The second statement is one of the specific SELECT statements that are used to retrieve the records that contain the answer trie. It takes no input parameters and returns only one field, ARG1. Note that value 22 is the YapTab's internal representation of the integer term of value 1. The third statement presents a similar query that will retrieve answers if floating-point values are expected to unify with the variable term Y. Finally, the forth statement collects the meta-data for the subgoal.

```
(1) INSERT IGNORE INTO SESSIONk_F2(META,ARG0,ARG1) VALUES(?,?,?);
(2) SELECT ARG1 FROM SESSIONk_F2 WHERE META=0 AND ARG2=22;
(3) SELECT F2.ARG1, FLOATS.VALUE AS FLT_ARG1
    FROM   SESSIONk_F2 AS F2 LEFT JOIN SESSIONk_FLOATS AS FLOATS
           ON (F2.ARG1=FLOATS.TERM)
    WHERE META=0 AND ARG2=22;
(4) SELECT ARG1 FROM SESSIONk_F2 WHERE META=1 AND ARG2=22;
```

Fig. 4. Prepared statements for $f(Y,1)$

3.4 The DBTAB API

We next present the list of developed functions and briefly describe their actions.

`dbtab_init_session(int sid)` - initializes the session passed by argument;

`dbtab_kill_session(void)` - kills the currently opened session;

`dbtab_init_table(TableEntry tab_ent)` - creates the relational table and initializes the generic INSERT prepared statement associated with the table entry passed by argument;

`dbtab_free_table(TableEntry tab_ent)` - frees the INSERT prepared statement, dropping the table if no other instance is using it;

`dbtab_init_view(SubgoalFrame sg_fr)` - initializes the specific SELECT prepared statement associated with the passed subgoal frame;

`dbtab_free_view(SubgoalFrame sg_fr)` - frees the SELECT prepared statement;

`dbtab_export(SubgoalFrame sg_fr)` - traverses both the subgoal trie and the answer trie, executing the INSERT prepared statement placed at the table entry associated with the subgoal frame passed by argument. The answer trie is deleted at the end of the transaction;

`dbtab_import(SubgoalFrame sg_fr)` - starts a data retrieval transaction, executing the SELECT prepared statement for the subgoal frame passed as argument.

4 Extending the YapTab Design

When a predicate is declared as tabled, the `dbtab_init_table()` function is called, starting the table creation process and generation of the INSERT clause,

sending it afterwards to the database for parsing. If preparation succeeds, the returned handle is placed inside the corresponding table entry data structure.

DBTAB's final model is meant to trigger the dumping of a tabled subgoal to the database when the corresponding table is chosen by YapTab's memory management algorithm to be abolished. Currently, DBTAB is still not yet fully integrated with YapTab's memory management algorithm. However, the present version already implements all the required features to correctly export and import tables, therefore allowing us to study and evaluate the potential and weaknesses of the proposed model. The current version of DBTAB triggers the dumping of a tabled subgoal to the record-set upon its *completion* operation, removing it from memory afterwards - it is to be replaced by a record-set storing the same answer terms. This operation is delayed up to this point in execution in order to prevent unnecessary memory consumption, both at client and server sides, while only incomplete tables are known. Variant calls to completed subgoals always import answers from the database.

4.1 Exporting Answers

Figure 5 shows the pseudo-code for the `dbtab_export()` function. Initially, the function starts a new data transaction. It then begins to climb the subgoal trie branch, binding the ground terms to the respective statement parameters along the way. When the root node is reached, all parameters consisting of

```
dbtab_export(SubgoalFrame sg_fr) {
  dbtab_start_transaction()
  insert_stmt = TabEnt_insert_stmt(SgFr_tab_ent(sg_fr))
  n_vars = bind_subgoal_terms(SgFr_parent(sg_fr))
  answer = SgFr_first_answer(sg_fr)
  while (answer != NULL) {
    bind_answer_terms(answer)                            // prepare record
    commit &= exec_prep_stmt(insert_stmt)
    answer = TrNode_child(answer)
  }
  if (!n_vars) {                // n_vars is the number of free variables
    bind_subgoal_metadata(n_vars)                   // prepare meta-record
    commit &= exec_prep_stmt(insert_stmt)
  }
  if (commit) {
    mysql_commit(DBTAB_SCHEMA)
    mark_as_stored(sg_fr)                   // update subgoal frame state
    free_answer_trie(SgFr_answers(sg_fr))
  } else {
    mysql_rollback(DBTAB_SCHEMA)
  }
}
```

Fig. 5. Pseudo-code for `dbtab_export()`

variable terms will be left NULL. The attention is then turned to the answer trie, cycling through the terms stored within the answer nodes. The remaining NULL parameters are bound repeatedly, and the prepared statement is executed for each present branch. Next, the meta-information about variables is stored. For each variable term present in the subgoal trie branch, a new unassigned variable term is created. The non-tag part of this variable is used to store a bit-mask containing information about all the possible types of terms that will be unified with the original variable. The total number of variables is stored in the META field of this record. Finally, the COMMIT of the transaction occurs if and only if all INSERT statements are executed correctly; otherwise, a ROLLBACK operation is performed.

To clarify ideas, recall the example of Fig. 1. During the execution of the '`:- table f/2.`' directive, table SESSIONk_F2 is created in the DBTAB database and the INSERT statement, meant to handle all insertions into this table, is generated and sent to the database, which will return a handle for it upon successful parsing. The handle is placed inside a prepared statement data structure pointed by the `TabEnt_insert_stmt` field of the table_entry data structure (see Fig. 6 for details). The structure's field `stmt_buff` and `params` sub-structure are initialized, with `params.bind` being set to point at a newly created array of MYSQL_BIND structures.

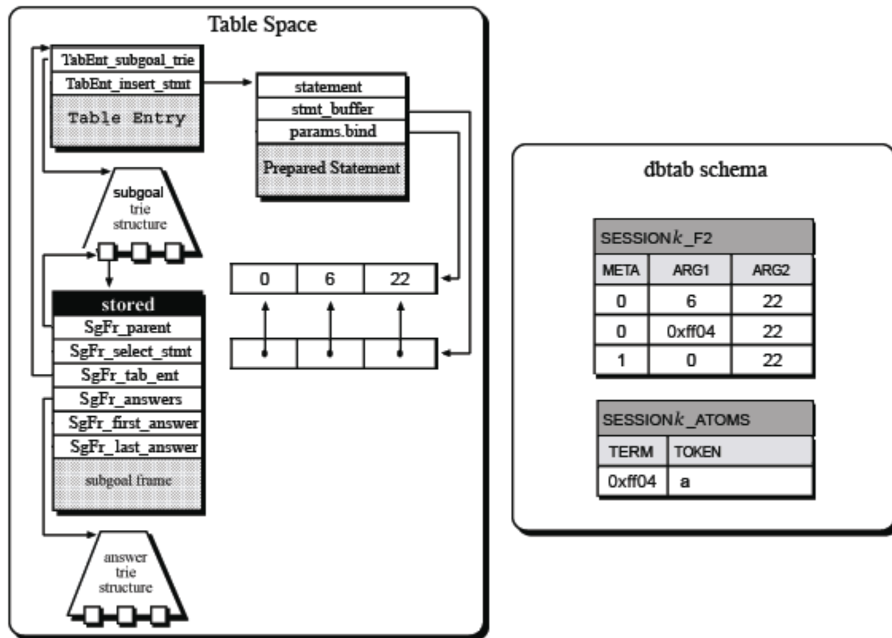


Fig. 6. Exporting $f(Y, 1)$: the relational representation

When completion is reached, the subgoal trie is climbed binding the second parameter, ARG2, with the integer term of value 1 (appearing in its internal representation 22). All values for ARG1 are then bound cycling through the leafs

of the answer trie. Each branch is climbed up to the root node, that marks the point where the insertion is to be performed. The branch for the integer term of value 0 (internally represented by 6) is stored right away, but the one for atom a (internally represented by 0xff04 in the figure) requires extra work because atoms are also stored into the corresponding SESSIONk_ATOMS table. At last, the meta-data is inserted. This consists of a record holding the different terms found in the answer trie for the free arguments in the subgoal call along with the other ground arguments. A new variable term replaces $VAR0$, and its non-tag part is used to hold a bit-mask - the value 0 visible in the last record - signaling that ARG1 column holds no special typed terms[5]. The META field holds the number of free variables for the subgoal, 1 in this case.

4.2 Importing Answers

After completion, the first variant call to a stored subgoal call now executes the `dbtab_import()` function, presented at Fig. 7.

```
dbtab_import(SubgoalFrame sg_fr) {
  select_stmt = SgFr_select_stmt(sg_fr)
  if (!PS_STMT(select_stmt)) {
    dbtab_init_view(sg_fr)
    exec_prep_stmt(select_stmt)
  }
  // switch on the number of rows
  if (PS_NROW(select_stmt) == 0) {                           // no answers
    SgFr_first_answer(sg_fr) = NULL
    SgFr_last_answer(sg_fr) = NULL
    SgFr_answers(sg_fr) = NULL
  } else {
    SgFr_first_answer(sg_fr) = PS_TOP_RECORD(select_stmt)
    SgFr_last_answer(sg_fr) = PS_BOTTOM_RECORD(select_stmt)
    if(VIEW_FIELD_SUCCEED == TRUE)                           // yes answer
      SgFr_answers(sg_fr) = NULL
    else                                            // one or more answers
      SgFr_answers(sg_fr) = SgFr_first_answer(sg_fr)
  }
}
```

Fig. 7. Pseudo-code for `dbtab_import()`

The first step calls the `dbtab_init_view()` function, creating the SELECT prepared statement that will load all possible answers. All variable terms are returned by default, possibly in conjunction with additional columns for non-atomic values. In case floating-point values are to be returned by ARGk, an additional column FLT_ARGk is joined to the data-set (see query 3 at Fig. 4 for such an example). Likewise, if long integers are to be returned by ARGk,

[5] Meaning floating-point numbers or long integers.

an additional column LINT_ARGk is joined to the data-set. These columns are placed immediately to the right of ARGk and possibly both of them may appear simultaneously - if such is the case, only one of these columns is set to a non-NULL value. The choice on which one of them to use is made consulting ARGk, who's value is replaced by a functor term for the desired type[6].

Also during this process, ground terms are used to set search conditions, within the WHERE clause, to be matched upon data retrieval in order to shorten the fields list, thus reducing the amount of data returned by the server. The statement is finally sent to the database for parsing and, on success, the returned handle is stored inside the prepared statement data structure added to the subgoal frame.

Since the predicate's answer trie will not change once completed, all subsequent calls may fetch their answers from the obtained record-set. The next step is then to reset the subgoal frame `SgFr_first_answer`, `SgFr_last_answer` and `SgFr_answers` internal fields accordingly to the obtained data-set:

Ground queries return at most one record. On failure, the pointers are all set to NULL and no record is returned, which means that the answer is *no*. On success, `SgFr_first_answer` and `SgFr_last_answer` point at the only record of the fetched data-set, consisting of a single boolean field named SUCCEED holding a TRUE value, and `SgFr_answers` holds the NULL value, indicating this is a *yes* answer.

Non-ground queries may return more than one record. If the reduction of the subgoal holds, the `SgFr_answers` and `SgFr_first_answer` pointers are set respectively to the first record of the data-set, while `SgFr_last_answer` is set to the last.

Figure 8 shows answer collection for $f(Y, 1)$. The meta-data is recovered through the execution of query 4 at Fig. 4. The constant term 1 (internally represented by value 22) is used to set a search condition over ARG2. All values in column ARG1 are then recovered as possible answers for variable term Y through the execution of query 2 at Fig. 4. At last, the subgoal frame pointers `SgFr_answers`, `SgFr_first_answer` and `SgFr_last_answer` are set to the first and last records as explained above.

As control returns from `dbtab_import()`, the `SgFr_answers` value is tested to decide if the query should fail, proceed or load answers from the database. If loading answers, the first record's offset along with the subgoal frame address are stored within a *loader choice point*[7]. The fetched record and its field values are then used to bind the free variables found for the subgoal in hand. If backtracking occurs, the choice point is reloaded and its `CP_last_answer` field, containing the offset for the last consumed record, is used to calculate the offset for the next answer. If the new offset is a valid one, the `CP_last_answer` is updated

[6] YapTab defines internally two special functors for this purpose: `FunctorDouble` and `FunctorLongInt`.

[7] A loader choice point is a WAM choice point augmented with the offset for the last consumed record and a pointer to the subgoal frame data structure.

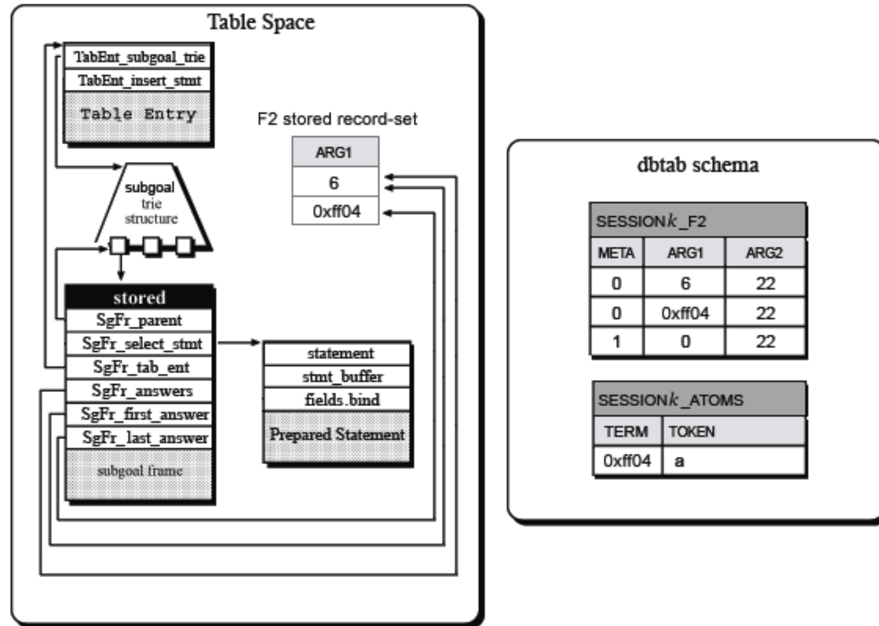


Fig. 8. Importing $f(Y, 1)$: the resulting data-set

accordingly. Otherwise, the choice point is discarded, signaling the positioning at the last answer. Whatever the case, the record is fetched and the variables are rebound according to the fields values. This process continues until all answers are consumed.

5 Initial Experimental Results

A batch of tests were performed in a computer with a Pentium®4 2.6GHz processor and 1GB of RAM. The test program, shown in Fig. 9, is a simple path discovery algorithm over a graph.

```
:- consult('graph.pl').
:- tabling_init_session(S).
:- table path/2.

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

Fig. 9. The test program

For comparison purposes, three main series of tests were performed both in YapTab and DBTAB environments. For each one of these series, the external file that holds the `edge/2` facts was generated with a different number of edges, ranging roughly from 50 to 150, corresponding to 5000 to 50000 possible combinations among nodes. In each sub-series, three types of nodes were considered:

integer, floating-point and atom terms. The query '`?- path(X,Y).`' was executed 10 times for each setup and the mean of measured times, in milliseconds, is presented in Table 1. The table shows two columns for YapTab, measuring the generation and recovery times when using tries to represent the table space, and three columns to DBTAB, measuring the times to export and import the respective number of answers and the time to recover answers when navigating the stored data-set after importing it.

Answers	Terms	YapTab		DBTAB		
		Generation	Recovery	Export	Import	Recovery
5000	integers	23	1	387	41	2
	atoms	21	2	1148	37	3
	floats	22	2	1404	54	3
10000	integers	58	2	780	60	3
	atoms	66	2	2285	63	4
	floats	64	3	2816	94	5
50000	integers	413	5	3682	240	15
	atoms	422	6	11356	252	12
	floats	386	20	14147	408	34

Table 1. Execution times, in milliseconds, for YapTab and DBTAB

As expected, most of DBTAB's execution time is spent in data transactions, mainly during insertion of tuples. Storage of non-integer terms takes approximately three times more than their integer counter-part, due to the extra insertion on auxiliary tables. An implementation of this step using stored procedures might accelerate things a little bit, since it takes only one message to be sent and most of the processing is done server-side. Non-atomic terms (floats) also present an interesting problem at fetching time. The use of LEFT JOIN clauses in the retrieval select statement (as seen in Fig. 4) becomes a heavy weight when dealing with large data-sets. Some query optimization is required to simplify the process and decrease the time required to import answers.

Two interesting facts emerge from the table. First, the navigation times for tries and data-sets are relatively similar, with stored data-sets requiring, on average, the double of time to be completely scanned. The second observed fact regards the time required to recompute answer tries for atomic terms (integers and atoms). When the answer trie becomes very large (the 50000 tuples rows), its computation requires more time, almost the double, than the fetching (import plus recovery) of its relational representation. DBTAB may thus become an interesting approach when the complexity of recalculating the answer trie largely exceeds the amount of time required to fetch the entire answer data-set.

An important side-effect of DBTAB is the attained gain in memory consumption. Recall that trie nodes are represented with four fields each, of which only one is used to hold a symbol, the others being used to hold the addresses of parent, child and sibling nodes (please refer to section 2). Since the relational representation dispenses the three pointers and focus on the symbol storage, the size of the memory block required to hold the answer trie can be reduced by a

factor of four. This is the worst possible scenario, in which all stored terms are integers or atoms. For floating-point numbers the reducing factor raises to eight because, although this type requires four trie nodes to be stored, one floating-point requires most often the size of two integers. For long integer terms, memory gains go up to twelve times: three nodes are used to store them in the trie.

6 Conclusions and Further Work

In this work, we have introduced the DBTAB model: a relational database model to represent and store tables externally in tabled logic programs. We discussed how to represent tables externally in database storage; how to handle atomic terms such as integers, atoms and floating-point numbers; and how we have extended the YapTab tabling system to provide engine support for exporting and importing answers to and from the database.

DBTAB was designed to be used as an alternative approach to the problem of recovering space when the tabling system runs out of memory. The common control implemented in most tabling systems is to have a set of tabling primitives that the programmer can use to dynamically delete some of the tables. By storing tables externally instead of deleting them, DBTAB avoids re-computation when subsequent calls to those tables appear. Another important aspect of DBTAB is the gain in memory consumption when representing answers for floating-point and long integer terms. Our preliminaries results showed that DBTAB may become an interesting approach when the cost of recalculating a table largely exceeds the amount of time required to fetch the entire answer data-set from the database.

As further work we plan to investigate the impact of applying DBTAB to a more representative set of programs. We also plan to introduce some other enhancements to improve the quality of the developed model. The expansion of the actual DBTAB model to cover all possibilities for tabling presented by YapTab is the first goal to achieve in a near future. First implementation tests shown that pairs, lists and application terms can be recorded and recovered through the use of a recursive algorithm and record trees. Each of these types of term is represented by a sequential number, which serves as the tree root, that is to be stored as described before at the tabled predicate relational table. Auxiliary tables have to be built to store all internal terms used by complex terms. These tables must possess a key field that links every node descendant to their direct ancestor. This operation is easy to implement and is expected to execute very quickly. Recovering is slightly more expensive. All child-nodes of the root node have to be selected, each one of them being interpreted as the root of a new sub-tree. The process continues until all leave-nodes are reached. By then, the specific term can be reconstructed by YapTab.

During execution, YapTab processes may have to stop due to several reasons: hosts may crash or have to be turned off, the users may want to interrupt process evaluation, etc. If such a situation arises, table space residing in memory is lost, leading to repeated calculation of the completed answer tries in later program ex-

ecutions. A possible solution to this problem is to search for meta-representation of terms before starting the process of tabling. If such a representation is found, the information contained in it can be used to not only build the corresponding branch in the subgoal tree but also the required prepared statements used to store new found answers and retrieve previously computed ones.

Acknowledgments

This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Pluri-anual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
4. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
5. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
6. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
7. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
8. Widenius, M., Axmark, D.: MySQL Reference Manual: Documentation from the Source. O'Reilly Community Press (2002)
9. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74

Linear Logic: Foundations, Applications and Implementations

Lukas Chrpa

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague
chrpa@kti.mff.cuni.cz

Abstract. Linear Logic is a powerful formalism used to manage a lot of problems with a finite number of resources. The most important feature of Linear Logic is its connectivity to Petri Nets. Linear Logic is also a good formalism which can be used in encoding planning problems. Linear Logic Programming is an extension of 'classical' logic programming and there exist several Linear Logic Programming languages that can solve problems encoded in Linear Logic more efficiently than 'classical' logic programming languages (Prolog). However existing Linear Logic Programming languages are not strong enough to solve some problems (for example planning problems). I believe that the best approach how to solve these problems is emulating Linear Logic in Prolog, but we need much more research in this area. In this paper I will describe Linear Logic, how to encode problems in Linear Logic, connection to Petri Nets and planning problems, Linear Logic Programming and my future research in this area.

1 Introduction

Linear Logic is a powerful formalism which can be used to formalize problems with a limited number of resources. In the real world we have many problems that have a limited number of resources. These problems can be also formalized with 'classical' logic, but the main disadvantage of 'classical' logic is a possible exponential growth of the length of formulas in face of the length of these problems.

Previous research gave us some interesting results, but many of these results are in a theoretical area. The most important result is a connectivity of Linear Logic with the Petri Nets. In my opinion, this result is very important, because we can see that Linear Logic can easily model more complicated problems. Another branch of research which leads from theory to practice is Linear Logic Programming. Using Linear Logic Programming we are able to solve problems based on Linear Logic much faster than using 'classical' Logic Programming. Linear Logic can also handle well planning problems. There exists a planner called RAPS [10] which is based on Linear Logic and its comparison to the best planners that participated at IPC (International Planning Competition) 2002

showed very interesting results. The computation time in computing the plans (typed Depots domains) was almost the best.

This paper will give the description of Linear Logic, the encoding of problems in Linear Logic (especially Petri Nets and planning problems), advantages and disadvantages of Linear Logic Programming, possibility of emulating Linear Logic in Prolog and the possible directions of my future research in this area. The main contribution of this paper is the overview of Linear Logic (especially the encoding of the planning problems which also belongs to my research interests) and possibility of emulation of Linear Logic in Prolog.

2 Linear Logic

Linear Logic was introduced by J.Y. Girard at 1987 [4,5,11]. As I mentioned in the introduction, Linear Logic was designed as a formalism which can handle with a finite number of resources. The main difference between 'classical' logic and Linear Logic is following: From A, A imply B we obtain B, but in 'classical' logic A is still available unlike in Linear Logic where A is no longer available. This is Girard's basic idea of Linear Logic. In the real world we often obtain resources by spending other resources. For example when we are in a shop buying some food, we are spending money on it.

2.1 Operators in Linear Logic

In the below description, predicates A and B mean resources.[1]

implication $A \multimap B$ — A basic operator in Linear Logic which means that B is obtained by spending (one) A.

multiplicative conjunction $A \otimes B$ — This operator means that A and B are used together. (If it's on the left side of implication both A and B are consumed, if it's on the right side of implication both A and B are obtained.)

multiplicative disjunction $A \wp B$ — This operator means that 'if not A than B' ($A^{\perp} \multimap B$).

additive conjunction $A \& B$ — This operator means that we have a choice between A and B. The choice depends on user.

additive disjunction $A \oplus B$ — This operator means that someone else has a choice between A and B.

exponential $!A$ — This operator converts A to a non-linear resource which means that we have an unlimited number of A. This operator provides a 'bridge' between 'classical' and Linear Logic.

exponential $?A$ — This operator means that we have an unknown amount of A.

negation $A^{\perp}$ — This operator provides duality between multiplicative operators ($\otimes, \wp$), additive operators ($\&, \oplus$) and exponentials (!,?). Double negation provides identity ($(A^{\perp})^{\perp} = A$).

[1] In the next sections will be used a notion 'linear facts' which may point out that Linear Logic predicates (resources) are used.

For the practice purpose we need only the following operators: $\multimap, \otimes, \&, \oplus, !$. The other operators are practically almost unusable. Syntax of Linear Logic has four constants ($1, \top, \bot, 0$) that are defined as neutral elements of the operators ($\otimes, \&, \wp, \oplus$).

Now I will describe the operators ($\multimap, \otimes, \&, \oplus, !$) in more details. The meaning of the implication ($\multimap$) is described above. On the left side of implication the resources are spent and on the right side of implication the resources are obtained. Meaning of the other operators depends on which side of implication they occur.

Example 1.

$$A \otimes B \multimap C \otimes D$$

This expression means that C and D are obtained by spending both A and B.

Example 2.

$$A \otimes B \multimap C \oplus D$$

This expression means that C or D is obtained (not both) by spending both A and B, but we don't know which one. When this expression appears in proving, we must split the proof into two proofs, where first proof contains possibility that C is obtained ($A \otimes B \multimap C$) and second proof contains possibility that D is obtained ($A \otimes B \multimap D$).

Example 3.

$$A \multimap B \& C$$

This expression means that B or C is obtained (not both) by spending A and we can choose which one. When this expression appears in proving, we should split the expression (not proof) into two expressions ($A \multimap B$ and $A \multimap C$).

2.2 Sequent Calculus for Linear Logic

Sequent calculus notation for Linear Logic is based on Gentzen's style like the other kinds of logic. This notation uses roman letters for propositions and greek letters for sets of formulas. In Linear Logic the sequent $\Delta \vdash \Gamma$ means that the multiplicative conjunction ($\otimes$) of the formulas in Δ implies ($\multimap$) the multiplicative disjunction ($\wp$) of the formulas in Γ. Proving in Linear Logic is quite similar to proving in 'classical' logic and it has the following form:

$$\frac{\text{Hypothesis1} \qquad \text{Hypothesis2}}{\text{Conclusion}}$$

Sequent calculus rules for Linear Logic are defined in the following way (according to the previous subsection I removed all rules containing the operators $\wp, ?, ^{\bot}$):

Identity	$A \vdash A$	$\frac{\Delta_1 \vdash A, \Gamma_1 \qquad \Delta_2, A \vdash \Gamma_2}{\Delta_1, \Delta_2 \vdash \Gamma_1 \Gamma_2}$	**Cut**
Perm. Left	$\frac{\Delta_1, A, B, \Delta_2 \vdash \Gamma}{\Delta_1, B, A, \Delta_2 \vdash \Gamma}$	$\frac{\Delta \vdash \Gamma_1, A, B, \Gamma_2}{\Delta \vdash \Gamma_1, B, A, \Gamma_2}$	**Perm. Right**
$\otimes$ **Left**	$\frac{\Delta, A, B \vdash \Gamma}{\Delta, (A \otimes B) \vdash \Gamma}$	$\frac{\Delta_1 \vdash A, \Gamma_1 \qquad \Delta_2 \vdash B, \Gamma_2}{\Delta_1, \Delta_2 \vdash (A \otimes B), \Gamma_1 \Gamma_2}$	$\otimes$ **Right**
$\multimap$ **Left**	$\frac{\Delta_1 \vdash A, \Gamma_1 \qquad \Delta_2, B \vdash \Gamma_2}{\Delta_1, \Delta_2, (A \multimap B) \vdash \Gamma_1 \Gamma_2}$	$\frac{\Delta, A \vdash B, \Gamma}{\Delta \vdash (A \multimap B), \Gamma}$	$\multimap$ **Right**
& **Left**	$\frac{\Delta, A \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma} \quad \frac{\Delta, B \vdash \Gamma}{\Delta, (A \& B) \vdash \Gamma}$	$\frac{\Delta \vdash A, \Gamma \qquad \Delta \vdash B, \Gamma}{\Delta \vdash (A \& B), \Gamma}$	& **Right**
$\oplus$ **Left**	$\frac{\Delta, A \vdash \Gamma \qquad \Delta, B \vdash \Gamma}{\Delta, (A \oplus B) \vdash \Gamma}$	$\frac{\Delta \vdash A, \Gamma}{\Delta \vdash (A \oplus B), \Gamma} \quad \frac{\Delta \vdash B, \Gamma}{\Delta \vdash (A \oplus B), \Gamma}$	$\oplus$ **Right**
Weakening	$\frac{\Delta \vdash \Gamma}{\Delta, !A \vdash \Gamma}$	$\frac{\Delta, !A, !A \vdash \Gamma}{\Delta, !A \vdash \Gamma}$	**Contraction**
Dereliction	$\frac{\Delta, A \vdash \Gamma}{\Delta, !A \vdash \Gamma}$		
0 Left	$\Delta, 0 \vdash \Gamma$	$\Delta \vdash \top, \Gamma$	$\top$ **Right**
$\bot$ **Left**	$\bot \vdash$	$\frac{\Delta \vdash \Gamma}{\Delta \vdash \bot, \Gamma}$	$\bot$ **Right**
1 Left	$\frac{\Delta \vdash \Gamma}{\Delta, 1 \vdash \Gamma}$	$\vdash 1$	**1 Right**
Forall	$\frac{\Delta, A \vdash \Gamma}{\Delta, (\forall x) A \vdash \Gamma}$	$\frac{\Delta, A \vdash \Gamma}{\Delta, (\exists x) A \vdash \Gamma}$	**Exists**

The next example shows how the proving in Linear Logic works. Assume this expression:

$$A, !(A \multimap B) \vdash B$$

The proof of the expression:

$$\dfrac{\dfrac{A \vdash A \qquad B \vdash B}{A, A \multimap B \vdash B}\ \multimap \textbf{Left}}{A, !(A \multimap B) \vdash B}\ \textbf{Dereliction}$$

2.3 Decidability and complexity of Linear Logic

This subsection will give an overview of decidability and complexity of propositional Linear Logic and its fragments. The following table shows the complexity of fragments of Linear Logic:

Fragments	Complexity
$\multimap, \otimes, \wp, \&, \oplus, !, ?$	Undecidable [12]
$\multimap, \otimes, \wp, \&, \oplus$	PSPACE-complete [12]
$\multimap, \otimes, \wp$	NP-complete [8]
$\multimap, \otimes, \wp, !, ?$	Unknown

3 Encoding problems in Linear Logic

As I mentioned above Linear Logic is powerful in encoding problems with finite number of resources. In this section it will be shown the encoding of simple

problems, the encoding of Petri Nets and planning problems will be shown in following sections.

3.1 Hamiltonian cycle

From the theory of graphs we know that a Hamiltonian cycle is a cycle which contains all vertices. Assume directed graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. We know that in a Hamiltonian cycle every vertex is used just once, so we can represent these vertices as linear facts. Edges can be represented as 'classical' facts (in Linear Logic we can use exponential !). To find the Hamiltonian cycle we need this rule:

$$at(v_i) \otimes v_i \otimes e(v_i, v_j) \multimap at(v_j)$$

The predicate *at* is used as a marker describing the current vertex from which we continue searching for Hamiltonian cycle. The whole problem of Hamiltonian cycle can be converted to Linear Logic in the following way (the searching for Hamiltonian cycle begins at the vertex v_1):

$$v_1, \ldots, v_n, !e(v_{i_1}, v_{j_1}), \ldots, !e(v_{i_m}, v_{j_m}), at(v_1), !(at(v_i) \otimes v_i \otimes e(v_i, v_j) \multimap at(v_j)) \vdash at(v_1)$$

Hamiltonian cycle exists if and only if the previous expression is provable in Linear Logic.

3.2 Knight's tour

Knight's tour problem is a problem where we have a chessboard and a knight that must visit all the board fields exactly once. Possible knight's moves are showed in Fig.1. In this case the fields on the chessboard can be represented

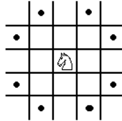


Fig. 1. Possible knight's moves

as linear facts. Assume that there exists a relation $next : (i, j) \rightarrow (\bar{i}, \bar{j})$ which computes the next move of the knight. The predicate *at* is used as a marker describing the current position of the knight. The problem of knight's tour can be converted to Linear Logic in the following way (the knight begins his tour at the field (1,1) and finishes his tour at the field (k, l)):

$$b(1, 1), \ldots, b(n, n), at(1, 1), !(at(i, j) \otimes b(i, j) \multimap at(next(i, j))) \vdash at(k, l) \otimes b(k, l)$$

Knight's tour problem has a solution if and only if the previous expression is provable in Linear Logic.

4 Connection between Linear Logic and Petri Nets

In this section we will describe how Petri Nets can be represented in Linear Logic.

Definition 1. *Petri Net is a 5-tuple $C = (P, T, I, O, \mu)$ where:*

- *P is a set of places.*
- *T is a set of transitions.*
- *$I : T \rightarrow P^{\infty}$ is an input function ($I(t), t \in T$ is a multi-set of input places of the transition t).*
- *$O : T \rightarrow P^{\infty}$ is an output function ($O(t), t \in T$ is a multi-set of output places of the transition t).*
- *$\mu : P \rightarrow N_0$ is a marking (marking is a vector describing the number of tokens in each place).*

Definition 2. *Assume Petri Net $C = (P, T, I, O, \mu)$.*

1. *The transition $t \in T$ is feasible if and only if $\forall p \in P, \quad \mu(p) \geq \#(p, I(t))$ is satisfied.*
2. *Assume that the transition $t \in T$ is feasible. After the execution of the transition t we obtain the new marking μ': $\mu'(p) = \mu(p) - \#(p, (I(t)) + \#(p, O(t)), \quad \forall p \in P$.*

In Petri Nets we often investigate whether some marking is reachable (by execution of a finite number of transitions) from the initial marking. More about this topic can be found in [15].

The encoding of Petri Nets to Linear Logic is quite simple. Tokens in places can be represented with linear facts, because the tokens can be perceived as resources. The transitions in the Petri Nets have a similar behavior like the implication in Linear Logic. The transition t can be encoded in Linear Logic in the following way ($\bigotimes A \ldots$ multiplicative conjunction of all elements from the multi-set A):

$$\bigotimes I(t) \multimap \bigotimes O(t)$$

Assume that $S(\mu)$ is a multi-set of places depending on the marking μ ($\forall p \in P : \#(p, S(\mu)) = \mu(p)$). The marking μ_g is reachable from marking μ_0 if and only if the following expression is provable in Linear Logic:

$$S(\mu_0), !\left(\bigotimes I(t_1) \multimap \bigotimes O(t_1)\right), \ldots, !\left(\bigotimes I(t_m) \multimap \bigotimes O(t_m)\right) \vdash \bigotimes S(\mu_g)$$

The coverage of the marking μ_g exists if and only if the following expression is provable in Linear Logic:

$$S(\mu_0), !\left(\bigotimes I(t_1) \multimap \bigotimes O(t_1)\right), \ldots, !\left(\bigotimes I(t_m) \multimap \bigotimes O(t_m)\right) \vdash \left(\bigotimes S(\mu_g)\right) \otimes \top$$

The difference between the previous expressions is only in the constant $\top$. Using the constant $\top$ likewise in the second expression changes the meaning of the proof from the reachability (we must have an exact number of tokens in places) to coverage (we must have at least the number of tokens in places).

4.1 Example

Assume the Petri Net from Fig. 2. In this case the multi-set $S(\mu_0)$ looks like:

$$S(\mu_0) = \{a, c, d, d\}$$

The encoding of the transitions t_1 and t_2:

$\mathbf{t}_1$ — $(a \otimes c) \multimap b$
$\mathbf{t}_2$ — $(b \otimes d \otimes d) \multimap (c \otimes d)$

To find out if the marking (one token in c and d) is reachable the following expression in Linear Logic must be provable:

$$a, c, d, d, !((a \otimes c) \multimap b), !((b \otimes d \otimes d) \multimap (c \otimes d)) \vdash c \otimes d$$

More about connection between Linear Logic and Petri Nets can be found at [14].

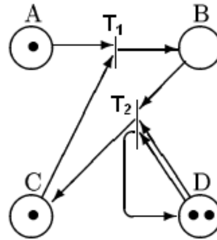


Fig. 2. Example of Petri Net

5 Planning in Linear Logic

Problem of using Linear Logic in planning problems has been studied by several researchers [13,9,2]. In this section we will present the connection between Linear Logic and planning problems (in this case a planning in the state space).

5.1 Preliminaries

Definition 3. *Assume that $L = \{p_1, \ldots, p_n\}$ is a finite set of predicates. Planning domain Σ over L is a 3-tuple (S, A, γ) where:*

- *$S \subseteq 2^L$ is a set of states. $s \in S$ is a state. If $p \in s$ then p is true in s and if $p \notin s$ then p is not true in s.*
- *A is a set of actions. Action $a \in A$ is a 3-tuple $(p(a), e^-(a), e^+(a)) \subseteq S$ where $p(a)$ is a precondition of the action a, $e^-(a)$ is a set of negative effects of the action a and $e^+(a)$ is a set of positive effects of the action a and $e^-(a) \cap e^+(a) = \emptyset$.*
- *$\gamma : S \times A \to S$ is a transition function. $\gamma(s, a) = (s - e^-(a)) \cup e^+(a)$ if $p(a) \subseteq s$.*

Definition 4. Planning problem P *is a 3-tuple (Σ, s_0, g) such that:*

- *$\Sigma = (S, A, \gamma)$ is a planning domain over L.*
- *$s_0 \in S$ is an initial state.*
- *$g \subseteq L$ is a set of goal predicates.*

Definition 5. Plan π *is an ordered sequence of actions $< a_1, \ldots, a_k >$ such that, the plan π* solves *the planning problem P if and only if $g \subseteq \gamma(\gamma(s_0, a_1), < a_2, \ldots, a_k >)$. Plan π is* optimal *if and only if for each π' $\mid \pi \mid \leq \mid \pi' \mid$ is valid.*

5.2 Basic encoding of planning problems

From the previous section we know that Petri Nets can be easily encoded by Linear Logic. This idea can be used in modeling the planning problems with Linear Logic. The predicates in the planning problems can be encoded as linear facts. Assume a state $s = \{p_1, p_2, \ldots, p_n\}$, its encoding in Linear Logic is following:

$$(p_1 \otimes p_2 \otimes \cdots \otimes p_n)$$

Assume an action $a = \{p, e^-, e^+\}$, its encoding in Linear Logic is following:

$$\forall i \in \{1, 2, \ldots, k\}, l_i \in p \cup e^-$$

$$\forall j \in \{1, 2, \ldots, m\}, r_j \in e^+ \cup (p - e^-)$$

$$(l_1 \otimes l_2 \otimes \cdots \otimes l_k) \multimap (r_1 \otimes r_2 \otimes \cdots \otimes r_m)$$

This expression means that the predicates on the left side of the implication will no longer be true after performing action a and the predicates on the right side of the implication will become true after performing action a. The whole planning problem can be encoded in Linear Logic in the following way (assume that $a_1, a_2 \ldots a_m$ are substitutions of the encodings of all the actions in A, $s_0 = \{p_{0_1}, p_{0_2}, \ldots, p_{0_n}\}$ and $g = \{g_1, g_2, \ldots, g_q\}$):

$$p_{0_1}, p_{0_2}, \ldots, p_{0_n}, !a_1, !a_2, \ldots !a_m \vdash g_1 \otimes g_2 \otimes \cdots \otimes g_q \otimes \top$$

A plan exists if and only if the previous expression is provable in Linear Logic.

5.3 Encoding of negative predicates

The previous encoding works with positive predicates only. However sometimes we need to encode negative predicates that obviously appear in preconditions of actions. We extend the encoding of predicates with symbols for negative predicates (predicate p will obtain a twin $\overline{p}$ which represents a negative form of predicate p). It stands to reson that either p or $\overline{p}$ is contained in each state (if $p \in s$ then the predicate p is true in the state s and if $\overline{p} \in s$ then the predicate p is not true in the state s). The encoding of the state s, where the predicates $p_1, \ldots, p_m$ are true in s and the predicates $p_{m+1}, \ldots, p_n$ are not true in s:

$$p_1 \otimes \cdots \otimes p_m \otimes \overline{p_{m+1}} \otimes \cdots \otimes \overline{p_n}$$

For each action $a = \{p, e^-, e^+\}$, we create an action $a' = \{p, e'^-, e'^+\}$, where $e'^- = e^- \cup \{\overline{p}\,|p \in e^+\}$ and $e'^+ = e^+ \cup \{\overline{p}\,|p \in e^-\}$. The whole planning problem with negative predicates can be encoded in Linear Logic in a similar way as the planning problem without negative predicates.

5.4 Encoding of possible optimizations

In the previous subsections I described the pure encoding of planning problems to Linear Logic. In this subsection I will show that we are able to encode some optimizations to Linear Logic as well.

Assume actions a and b such that $e^-(a) = e^+(b)$ and $e^+(a) = e^-(b)$. These actions are inverse which means that when performing them consecutively we obtain a state that we had before performing these actions. It stands to reson that performing these actions consecutively is unnecessary. The main idea to solve the problem of performing inverse actions consecutively is an extension of the encoding of actions. We can add a new predicate $can(a)$ which means that an action a can be performed ($can(a) \in p(a)$). To block the inverse action b by the action a, $can(b) \in e^-(a)$ and $\overline{can(b)} \in e^+(a)$ must be satisfied. To unblock the blocked action b by an action c, $can(b) \in e^+(c)$ and $\overline{can(b)} \in e^-(c)$ must be satisfied.

Another possible optimization is blocking some action forever. Blocking actions forever should be used especially in domain-depended planning (for example when we know that it is not necessary to move a box, we simply block the action that causes the moving of the particular box). From the previous paragraph we know that if $can(a) \notin s$ then the action a is blocked in the state s.

Another possible optimization is actions assembling. Assume that the actions a_1 and a_2 are encoded in Linear Logic in the following way:

$$\begin{aligned} a_1 : \quad & (p_1 \otimes p_2) \multimap (p_1 \otimes p_3) \\ a_2 : \quad & (p_3 \otimes p_4) \multimap (p_3 \otimes p_5) \end{aligned}$$

The encoding in Linear Logic of action a which is obtained by assembling the actions a_1, a_2:

$$(p_1 \otimes p_2 \otimes p_4) \multimap (p_1 \otimes p_3 \otimes p_5)$$

The above optimizations can rapidly increase the performance of a planner. This section showed that Linear Logic is strong enough to encode some optimizations (mainly domain-dependent) for the planning problems.

5.5 Example

In this example we will use a predicate extension of Linear Logic. Imagine the version of "Block World", where we have slots and boxes, and every slot may contain at most one box. We have also a crane, which may carry at most one box.

Initial state: 3 slots (1,2,3), 2 boxes (a, b), empty crane, box a in slot 1, box b in slot 2, slot 3 is free.

Actions:

$PICKUP(Box, Slot) = \{$

$$\begin{aligned} p &= \{empty, in(Box, Slot)\}, \\ e^- &= \{empty, in(Box, Slot)\}, \\ e^+ &= \{holding(Box), free(Slot)\} \\ &\} \end{aligned}$$

$PUTDOWN(Box, Slot) = \{$

$$\begin{aligned} p &= \{holding(Box), free(Slot)\}, \\ e^- &= \{holding(Box), free(Slot)\}, \\ e^+ &= \{empty, in(Box, Slot)\} \\ &\} \end{aligned}$$

Goal: Box a in slot 2, Box b in slot 1.

The encoding of the action $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$:

$PICKUP(Box, Slot)$:
$empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot)$

$PUTDOWN(Box, Slot)$:
$holding(Box) \otimes free(Slot) \multimap empty \otimes in(Box, Slot)$

The whole problem is encoded in Linear Logic in the following way:

$$in(a, 1), in(b, 2), free(3), empty, !PICKUP(X, Y), !PUTDOWN(X, Y) \vdash \\ \vdash in(b, 1) \otimes in(a, 2) \otimes \top$$

It stands to reson that the actions $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$ are inverse. The encoding of the action $PICKUP(Box, Slot)$ uses blocking of the inverse action $PUTDOWN(Box, Slot)$ and unblocking of the previously blocked action $PICKUP(X, Y)$ (the encoding of the action $PUTDOWN(Box, Slot)$ is

analogical):

$PICKUP(Box, Slot):$
$can(PICKUP(Box, Slot)) \otimes can(PUDOWN(Box, Slot)) \otimes \overline{can(PICKUP(X, Y))} \otimes$
$empty \otimes in(Box, Slot) \multimap holding(Box) \otimes free(Slot) \otimes can(PICKUP(Box, Slot)) \otimes$
$\overline{can(PUTDOWN(Box, Slot))} \otimes can(PICKUP(X, Y))$

Another possible optimization in this example is assembling the actions $PICKUP(Box, Slot)$ and $PUTDOWN(Box, Slot)$ into the action $MOVE(Box, SlotX, SlotY)$. The action $MOVE(Box, SlotX, SlotY)$ is encoded in the following way:

$MOVE(Box, SlotX, SlotY)$
$empty \otimes in(Box, SlotX) \otimes free(SlotY) \multimap empty \otimes in(Box, SlotY) \otimes free(SlotX)$

5.6 Using Linear Logic in planning under uncertainty

The main difference between the deterministic planning and the planning under uncertainty is such that the actions in the planning under uncertainty can reach more states. The main advantage of Linear Logic are additive operators. Difference between additive conjunction (&) and additive disjunction ($\oplus$) rests in the fact that usage of additive disjunction means that there is plan that certainly succeeds, and usage of additive conjunction means that there is some plan that should succeed (with nonzero probability). The encoding of the actions in the planning under uncertainty($s, s_1, s_2, \ldots, s_n$ are states, a is the action):

$$s \times a \rightarrow \{s_1, s_2, \ldots, s_n\}$$

$$a : s \multimap (s_1 \&(\oplus) s_2 \&(\oplus) \ldots \&(\oplus) s_n)$$

This expression means that only one state from $s_1, s_2, \ldots, s_n$, could be reached after performing the action a from the state s in a certain step.

6 Linear Logic Programming

Linear Logic Programming is derived from 'classical' logic programming by including linear facts and linear operators. Syntax of common Linear Logic Languages is based on Horn's clauses like in Prolog.

6.1 Linear Logic Programming languages

There exists several Linear Logic Programming languages. For example Lolli [7,6], LLP [1], Lygon [16], LTLL.[2] LTLL and LLP are possibly the most effective Linear Logic Programming languages today. The benchmarks showed that the Linear Logic Programming languages have much better efficiency in solving problems formalized in Linear Logic than Prolog.

[2] developed by Arnost Vecerka, more at http://www.inf.upol.cz/vecerka

The construction of interpreters of Linear Logic Programming languages is well described in [1,7]. The main disadvantage of these interpreters is the fact that the interpreters are not powerful enough to handle the linear implication ($\multimap$) well, because these interpreters are based on Horn clauses. In practice these interpreters can solve only the problems where the resources (linear facts) are spent. From the problems described in previous sections the interpreters of Linear Logic Programming languages can solve only Hamiltonian cycle and Knight's tour. For solving the Petri Nets reachability problems or the planning problems these interpreters are too weak.

6.2 Emulating Linear Logic in Prolog

In the previous subsection it was mentioned that the Linear Logic Programming languages are not strong enough to solve the planning problems. In my thesis [3] I proposed a Linear Logic Programming language SLLL, which was constructed as a compiler to Prolog. The main idea which I used is the fact that linear facts can be held in a special list. To keep the list of linear facts consistent during the computation we must add to each rule two parameters. The first parameter represents the list that inputs into the rule. The second parameter represents the list that outputs from the rule. The main advantage of this approach is easy implementation and vulnerability to backtracking. The main disadvantage of this approach is low computational efficiency.

I proposed the emulation of Linear Logic (only the operators $\otimes, \oplus, \multimap$ are needed) based on the special list of linear facts which I used in SLLL. Spending the linear fact is done when the linear fact is removed from the list. Obtaining the linear fact is done when the linear fact is added to the list. We must define two predicates, one for deleting the facts from the list (*lin_del*) and one for adding the facts to the list (*lin_add*):

```
lin_del(V,[V|L],L).
lin_del(V,[H|L],[H|NL]):-lin_del(V,L,NL).
lin_add(V,L,[V|L]).
```

The emulation of multiplicative conjunction ($\otimes$) is very easy, because we can replace them by 'classical' conjunction. The emulation of additive conjunction (&) and additive disjunction ($\oplus$) can be replaced by 'classical' disjunction, when additive conjunction (&) is on the right side of implication and additive disjunction ($\oplus$) is on the left side of implication.[3] Linear implication ($\multimap$) is emulated in such a way that all linear facts on the left side of the implication are deleted from the list and all linear facts on the right side of the implication are added to the list. A formula $a \otimes b \multimap c\&d$ can be written in Prolog in the following way:

```
(lin_del(a,L1,L2),lin_del(b,L2,L3)),
(lin_add(c,L3,L4);lin_add(d,L3,L4))
```

[3] If additive conjunction (disjunction) is used of the left (right) side of implication, the computation must be split into more branches and find a solution in each branch.

Variables $L1, L2, L3, L4$ represent the list of linear facts in the way mentioned in the first paragraph of this subsection.

The above approach can be modified by keeping the list of linear facts sorted. The advantage of this approach is such that if we are removing the linear facts from the list consecutively we need only one scan of the list. The disadvantage of this approach is such that to keep the list sorted, adding of linear facts to the list can't be done in a constant time. The predicates *lin_del* and *lin_add* are defined in the following way (I used a build-in predicate *merge* which merges two sorted lists):

```
lin_del([V],[V|L],L).
lin_del([V|VL],[V|L],NL) :- lin_del(VL,L,NL).
lin_del([V|VL],[H|L],[H|NL]) :- lin_del([V|VL],L,NL).
line_add(VL,L,NL) :- merge(L,VL,NL).
```

A formula $a \otimes b \otimes c \multimap d \otimes e$ can be written in the following way (both lists of predicates must be sorted):

```
(lin_del([a,b,c],L1,L2),lin_add([d,e],L2,L3))
```

However practical tests showed another problem that makes this approach less efficient that the previous one. The problem is such that if a linear fact which we want to remove can be unified with more linear facts in the list then we need to make more scans in the list that adds overhead. In the other hand, we are able to compare the sorted lists in linear time that can be used in detection of previously visited states. This approach has the best efficiency, but memory consumption is high.

7 Future research

In my future research, I will study the opportunities of Linear Logic and its efficient usage in encoding problems. I will also study the possibilities of usage Linear Logic Programming languages and their possible extensions and possibilities of efficient emulation of Linear Logic in Prolog. The following paragraphs will present my future research plans in more details:

7.1 Encoding problems and optimizations in Linear Logic

In previous sections I showed that some problems can be encoded in Linear Logic. In particular the planning problems can be well encoded in Linear Logic. However the pure encoding of the planning problems is not efficient enough. Using the optimizations that I mentioned greatly increases the efficiency. I will study more possibilities of using Linear Logic in the planning problems and I will try to find and encode more sophisticated optimizations.

The other problems that I did not mention are scheduling problems. In scheduling problems we are mapping some activities to resources under particular constraints. I believe that Linear Logic in strong enough to encode scheduling problems and I will study the possibilities of using Linear Logic in solving scheduling problems.

7.2 Using Linear Logic Programming languages

The main advantage of Linear Logic Programming languages is the high efficiency in solving problems based on Linear Logic. The main disadvantage of Linear Logic Programming languages is the impossibility to obtain the resources during the computation (we have the resources at the beginning of the computation and during the computation we can only spend these resources). As I mentioned in the previous section, Linear Logic Programming languages are not strong enough to solve the planning problems. In the other way I believe that Linear Logic Programming languages should be useful in solving scheduling problems or in some supporting techniques to the planning problems.

7.3 Emulating Linear Logic in Prolog

In the previous section I mentioned some possibilities how to emulate Linear Logic in Prolog. This emulation is correct and we are able to solve the planning problems with this emulation. However the efficiency of this emulation is still not very good. In the other way I believe that an efficient emulation of Linear Logic in Prolog should be very useful in developing efficient algorithms handling problems with limited resources. I will study the possibilities of more efficient emulations of Linear Logic in Prolog.

8 Conclusion

This paper showed that Linear Logic is a powerful formalism which can encode many ‘resource based‘ problems (for example the planning problems). The advantage of this approach is that an improvement of the Linear Logic solver leads to better efficiency in solving the problems encoded in Linear Logic. Unfortunately the existing Linear Logic solvers (Linear Logic Programming languages) are not strong enough to solve problems like planning problems. I believe that an efficient emulation of Linear Logic in Prolog will bring a lot of possibilities of usage Linear Logic in the problems.

9 Acknowledgements

The research is supported by the Czech Science Foundation under the contract no. 201/04/1102 and by the Grant Agency of Charles University (GAUK) under the contract no. 326/2006/A-INF/MFF.

References

1. Banbara, M. *Design and Implementation of Linear Logic Programming Languages.* Ph.D. Dissertation, The Graduate School of Science and Technology, Kobe University. 2002.

2. Chrpa L. Linear logic in planning. *To appear at Doctoral Consorcium ICAPS 2006.* 2006.
3. Chrpa, L. Linearni logika. Master's thesis, Department of Computer Science, Palacky University, Olomouc. 2005. (in Czech).
4. Girard J.-Y. Linear logic. *Theoretical computer science* 50:1–102. 1987.
5. Girard J.-Y. *Linear Logic: Its Syntax and Semantics.* Technical report, Cambridge University Press. 1995.
6. Hodas, J. Lolli: An extension of lambdaprolog with linear logic context management. *Proceedings of the 1992 Workshop on the lambdaProlog Programming Language.* 1992.
7. Hodas, J. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* Ph.D. Dissertation, University of Pennsylvania, Department of Computer and Information Science. 1994.
8. Kanovich M. The multiplicative fragment of linear logic is NP-complete. Technical Report X-91-13, Institute for Language, Logic and Information. 1991.
9. Kanovich, M., and Vauzeilles, J. The classical ai planning problems in the mirror of horn linear logic: Semantics, expressibility, complexity. *Mathematical Structures in Computer Science 11(6).* 2001.
10. Küngas P. Linear logic for domain-independent ai planning. *Proceedings of Doctoral Consorcium ICAPS 2003.* 2003.
11. Lincoln P. Linear logic. *Proceedings of SIGACT 1992.* 1992.
12. Lincoln P., Mitchell J., Scedrov M., and Shankar N. Decision problems for propositional linear logic. Technical Report SRI-CSL-90-08, CSL, SRI International. 1990.
13. Masseron, M.; Tollu, C.; and Vauzeilles, J. Generating plans in linear logic i-ii. *Theoretical Computer Science.* 1993.
14. Oliet, N. M., and Meseguer, J. From petri nets to linear logic. *Springer LNCS 389.* 1989.
15. Reisig, W. *Petri Nets, An Introduction.* Springer Verlag, Berlin. 1985.
16. Winikoff, M. Hitch hiker's guide to lygon 0.7. Technical Report 96/36, The University of Melbourne, Australia. 1996.