João Pedro Fernandes Raimundo

# Efficient Storing Mechanisms for Tabled Logic Programs


Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Outubro de 2010

**João Pedro Fernandes Raimundo**

# Efficient Storing Mechanisms for Tabled Logic Programs


*Tese submetida à Faculdade de Ciências da*
*Universidade do Porto para obtenção do grau de Mestre*
*em Ciência de Computadores*

*Orientador: Ricardo Jorge Gomes Lopes da Rocha*

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Outubro de 2010

# Acknowledgments

First of all, I must show my deepest gratitude to my supervisor, Professor Ricardo Rocha, for offering me the possibility to be part of his research team, giving me the chance to experience the world of scientific research. He guided me throughout all my questions, doubts and problems during this work, providing his ideas, knowledge and friendship to solve them. To him, I would like to thank for the revision and suggestions in the writing of this thesis. And also, for giving me the possibility to join the football guys and enjoying some relaxing and healthy moments.

I am also grateful to the STAMPA project, for the support with a research grant during my research work, and for the possibility to participate in international conferences.

To my old friends David Prino, Gonçalo Miranda and João Pereira, I would like to show my sincere gratitude for all the good moments from old and recent times, for everything.

To my friends from the world of Physics, André Pereira, Arlete Apolinário, Célia Sousa, João Ventura, João Amaral, Julie Berendt, Ricardo Silva and Sara Pinto, to some for the exquisite lunches and to others for their distinctive point of view of the reality, shared during this last years.

To my fellow co-workers of STAMPA project, for their support and friendship in all the long time spend working in our research room. And to all my friends from the Computer Science department for the relaxing moments at the department's bar and Fridays of football.

To my family, I would like to thank every one. In particular to my parents, Fernando and Cristina, for the chance to get a graduation, for all the support and for being so loving parents. To my brother for the special relation that we carry, as old as he is.

To my love, Diana Leitão, for all the support, guidance, advices (that sometimes, I stubbornly did not hear) and friendship. I will never thank you enough.

# Abstract

Programming languages are an unique method to communicate with machines. Declarative languages, such as logic programming languages, provide features like a high-level and declarative syntax, simplifying the communication between man-and-machine. Arguably, Prolog is the most famous and used logic programming language. Prolog uses SLD resolution in order to provide good performance in the computation of complex real world problems. Although SLD resolution proved to be very effective, in some cases, this procedure show some restrictions when dealing with infinite loops and redundant sub-computations.

One of the most successful techniques proposed to overcome SLD's susceptibility, is tabling. The tabling mechanism consists in storing the subgoals and the respective answers of a program in a table space in such a way that, in later stages of a program's evaluation, repeated subgoal calls use the answers stored in the tables, avoiding the subgoal re-evaluation. Tabling success largely depends on the implementation of the table space, its data structures and algorithms. Arguably, the most successful data structure for tabling is tries. Nevertheless, when tabling is used in applications that have large quantities of data, it can lead to overgrown tables and quickly fill up the system's memory.

With this research, we try to provide alternative designs and structures, not only to the table space organization but also to the tabled data representation. We do so, by proposing a new design for the table space organization where all terms in tabled subgoal calls and tabled answers are represented only once in a common global trie instead of being spread over several different trie data structures, suggesting three different approaches. At tabled data representation, we propose a new representation of list terms for tries that avoids the recursive nature of the WAM representation of list terms in which tries are based.

The results obtained in our experiments when using the YapTab tabling system, show significant reductions on memory usage, without compromising running time. Memory usage is reduced when using any of the three different global trie designs and also in the new representation of list terms, providing the necessary data to make it clear that our proposals can provide more compact and efficient representations of the table space, when applying tabling mechanisms to Prolog.

# Resumo

As linguagens de programação são um modo único de se comunicar com máquinas. Em particular, as linguagens declarativas, como são as linguagens de programação em lógica, oferecem uma sintaxe declarativa de alto nível, facilitando assim a comunicação entre homem e máquina. Indiscutivelmente, o Prolog é a linguagem de programação em lógica mais famosa e amplamente utilizada, usando a resolução SLD para proporcionar um bom desempenho no cálculo de problemas complexos do mundo real. Apesar da resolução SLD se ter mostrado muito eficaz, em alguns casos, este procedimento demonstrou algumas restrições, em particular quando se lida com ciclos infinitos e sub-computações redundantes.

Uma das técnicas propostas para superar as susceptibilidades da resolução SLD, é a tabulação. O mecanismo de tabulação consiste em guardar os subgolos de um programa e as respectivas soluções num espaço de tabelas de modo a que, durante a avaliação de um programa, quando acontece uma chamada repetida a um subgolo, são utilizadas as soluções já tabeladas, evitando assim que o subgolo seja reavaliado. O sucesso da tabulação depende em grande medida da implementação do espaço de tabelas, das suas estruturas de dados e algoritmos. Possivelmente, a mais bem sucedida estrutura de dados para a tabulação são as *tries*. No entanto, quando esta técnica é utilizada para tabelar soluções em aplicações com grande quantidade de dados, pode acontecer um crescimento desmesurado das tabelas, saturando rapidamente a memória do sistema.

Neste trabalho, apresentamos novas estruturas de dados alternativas, não só relacionadas com a organização do espaço de tabelas, mas também com a representação dos dados nelas representados. Fazêmo-lo, propondo um novo modelo para a organização do espaço de tabelas onde todos os subgolos tabelados e respectivas respostas são representados apenas uma vez numa *trie global*, em vez de serem distribuídos por várias tries diferentes, e para isso, sugerimos três abordagens distintas. Na representação dos dados tabelados, propomos uma nova representação dos termos lista nas *tries*, evitando a natureza recursiva da representação WAM para termos lista em que estas se baseiam.

Os resultados obtidos utilizando o sistema de tabulação YapTab, mostram uma redução significativa na utilização de memória, sem comprometer o tempo de execução. O uso de

memória é reduzido, quer seja ao utilizar qualquer uma das três abordagens com recurso a uma *trie global*, quer seja na utilização da nova representação dos termos lista, sugerindo que as nossas propostas conseguem uma representação mais compacta e eficiente do espaço de tabelas na utilização do mecanismo de tabulação em Prolog.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Logic programming languages, provide a high-level approach to programming. Noticeable, Prolog is the most used logic programming language. In fact, Prolog has proved to be very effective in application areas such as Artificial Intelligence, Natural Language Processing and Database Management to site just a few. Most of Prolog's success is in part due to David H. D. Warren's work, on the implementation of the WAM compiler to Prolog [1], providing a very efficient abstract machine for the implementation of Prolog systems [2].

Logic programming languages, such as Prolog, are based on Horn Clauses [3], a subset of first order logic. In fact, logic programs consist of a set of clauses, that provide the ground knowledge of programs. The execution of logic programs is reduced to query symbols manipulation until a refutation is found. This refutation, in Prolog, is provided by the SLD resolution [4] and done over Horn clauses for programs execution basis. Although, its proved power and declarativeness, SLD resolution can suffer from some limitations when dealing with infinite loops and redundant sub-computations. A proposal to solve those limitations is *tabling* [5, 6] which proved its viability due to the XSB Prolog system's work in the implementation of the SLG-WAM engine [7]. As a result, several different implementations of tabling mechanisms were developed and implemented in different Prolog systems. Examples of the variety of implementations of tabling are available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury and Ciao Prolog.

In a nutshell, tabling consists in storing intermediate answers for subgoals so that they can be reused whenever a repeated call appears. The performance of tabled evaluation largely depends on the implementation of the table space. In order to obtain an efficient response to systematic calls, fast lookup 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* [8]. Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms

and permits lookup and possibly insertion to be performed in a single pass through a term, hence resulting in a very efficient and compact data structure for term representation. When used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. A possible solution for this problem is to dynamically abolish some of the tables. This can be done using explicit tabling primitives or using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [9]. An alternative approach is to store tables externally in a relational database management system and then reload them back only when necessary [10]. A complementary approach to the previous problem is to study how less redundant, more compact and more efficient data structures can be used to better represent the table space. While tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated answers for different calls. The development of our work takes in consideration this last approach.

When representing terms in the trie, most tabling engines, like XSB Prolog, Yap Prolog and others, try to mimic the WAM [11] representation of these terms in the Prolog stacks in order to avoid unnecessary transformations when storing/loading these terms to/from the trie. Despite this idea seems straightforward for almost all type of terms, we found that this is not the case for list terms (also known as pair terms) and that, for list terms, we can design even more compact and efficient representations. In Prolog, a non-empty list term is formed by two sub-terms, the head of the list, which can be any Prolog term, and the tail of the list, which can be either a non-empty list (formed itself by a head and a tail) or the empty list. WAM based implementations explore this recursive nature of list terms to design a very simple representation at the engine level that allows for very robust implementations of key features of the WAM, like the unification algorithm, when manipulating list terms. However, when representing terms in the trie, the recursive nature of the WAM representation of list terms is negligible as we are most interested in having a compact representation with fast lookup and insertion capabilities.

## 1.1 Thesis Purpose

In this thesis, we present new proposals to the table space data structures and organization in order to improve the compactness and efficiency of tabled logic programs. We propose modifications, in a more comprehensive plan, to the table space representation and, in a amplified plan, to the structure of list terms representation.

Regarding the table space representation, we propose a new design and we, introduce three different approaches that are based in the usage of a common global trie. In all these three approaches, the representation of all tabled subgoal calls and/or answers is stored

in a common global trie instead of being spread over several different trie data structures. Our approaches resemble the *hash-consing* technique [12], as they try to share data that is structurally equal. An obvious goal is to save memory usage by reducing redundancy in the representation of tabled calls/answers to a minimum. Our first approach consists on storing subgoal call and answers in the global trie, thus reducing the number of nodes used in the subgoal and answer tries, and providing the possibility of reusing calls and answers already represented in the global trie. The second design maintains the use of a global trie, but only individual terms are represented in it. This increases the number of nodes in the original subgoal and answer tries but, on the other hand, also increases the reuse of the terms represented in the global trie. In the last approach, we once more use a global trie to store only terms, but as an alternative design we also try to maximize the reuse of individual terms present in the table space, by representing subterms (compound term's arguments) as unique entries in the global trie.

We also propose a new representation of list terms for tabled data that avoids the recursive nature of the WAM representation of list terms. In our new proposal, a list term is simply represented as an ordered sequence of the term elements in the list, i.e., we only represent the head terms in the sub-lists and avoid representing the sub-lists' tails themselves. Our experimental results show a significant reduction in the memory usage for the trie data structures and considerable gains in the running time for storing and loading list terms with and without compiled tries [13].

To implement these proposals, we will focus our work on a concrete implementation, the YapTab system [14, 15], but our proposals can be easy generalized and applied to other tabling systems.

## 1.2 Thesis Outline

The thesis is structured in seven chapters that can be seen as the representation of the different stages of our work. We provide next, a brief description of each chapter.

**Chapter 1: Introduction.** Is this chapter.

**Chapter 2: Logic Programming and Tabling.** Provides a brief overview of Logic Programming and the Tabling technique. Throughout, we discuss logic programming languages and abstract machines, focusing in Prolog and in the WAM, and also the mechanisms associated with the tabling technique, namely tabled evaluation and tries.

**Chapter 3: List Terms Representation.** First, it makes an introduction to YapTab's design for the representation of list terms and then, it presents our new and alternative design for list term representation, which the main goal is to optimize

YapTab's memory usage in order to reduce possible drawbacks of the standard mechanism.

**Chapter 4: Global Trie.** Presents the Global Trie (GT) design, specifying the three developed approaches to an alternative table space representation. The GT table space design emerges with the intent to surpass some of the disadvantages shown by YapTab standard table space design when dealing with redundant data, namely by storing terms in the same trie, thus preventing repeated representations of a term in different trie data structures.

**Chapter 5: Implementation.** In this chapter, we focus on the implementation details for the alternative table designs by describing the GT data structures and algorithms in more detail. Throughout, we also describe how tries are structured, specifying the main features of trie nodes, and present the main procedures which interact with tries, preforming comparisons with YapTab's original table design.

**Chapter 6: Experimental Results.** Presents experimental results comparing the new table space against the YapTab standard representation and discusses the obtained results.

**Chapter 7: Conclusions and Further Work.** Summarizes the work presented in the previous chapters, the reasons for the obtained results, and provides some guidelines for further work.

# Chapter 2

# Logic Programming and Tabling

This chapter provides a brief overview of the research areas comprehended in this thesis. We introduce the path from the general ideas of Logic Programming to the specifics of the Tabling technique. Throughout, we discuss logic programming languages and abstract machines, focusing in Prolog and in the WAM, and also the mechanisms associated with the tabling technique, namely tabled evaluation and tries.

## 2.1 Logic Programming

Programming languages are essential in making the communication between man-and-machine possible. The evolution of programming languages led to human-inspired languages, with syntaxes that appear more comprehensible and comparable to human writing. This particular kind of programming languages are called high-level. The declarative languages are a wide class of programming languages with the unique features of having a high-level language syntax. This class of languages are more concerned with the aspects of the problem that needs to be solved, instead of the actual method to solve it. Included in the declarative programming languages class, one has also logic and the functional languages. While the latter are based on $\lambda$-*calculus*, the former are completely different, relying on a subset of first-order logic and its procedural interpretation. Never the less by being based on formalisation of human thought, logic programming languages are arguably the more effective and straightforward way to allow programmers to easily express their reasoning.

Logic programming languages are based on a well known subset of first order logic, namely the Horn Clause [3]. Horn clauses contain a basic rule: at most one disjunct in the conclusion is required, meaning that at most one positive literal is needed. With this basic rule a Horn clause can be defined in three different forms:

- **Rule**, a clause that contains a positive literal and one or more negated literals. The most common form of a rule is

$$\neg q \vee ... \vee \neg r \vee \neg s \vee t$$

  and can also be written as,

$$t \leftarrow q \wedge ... \wedge r \wedge s$$

- **Fact**, when there are no negations and the clause is composed only with the positive literal, we have

$$t \leftarrow$$

- **Goal**, occurs when there is no positive literal

$$\leftarrow q \wedge ... \wedge r \wedge s$$

Logic programming languages show a syntactic equivalency to Horn clauses with minor changes. In logic programs the equivalent to a **rule** of the form

$$B \leftarrow A_1 \wedge ... \wedge A_{n-1} \wedge A_n$$

is (in the Prolog syntax) given by

$$B :- A_1, ..., A_{n-1}, A_n.$$

Additionally, one also finds other examples for Horn clauses, such as a **fact**

$$B.$$

and a **goal**

$$:- A_1, ..., A_{n-1}, A_n.$$

In this syntax, $B$ is the head of the clause and $A_1$ to $A_n$ are the body. Each B defines or is part of a predicate. Predicates exhibit the following form $p(t_1, ..., t_n)$, where the *t's* can be terms, and each term may have different representations. A simple term includes atoms or variables, while compound terms are specifically functors or lists. A functor is defined as $f(t_1, ..., t_n)$, where $f$ is the name of the functor and each $t$ represents different terms. A list is represented as $[t_1, ..., t_n]$, differing from functors by having no name associated.

In fact, logic programs are a set of clauses that form the ground knowledge of programs. To get results from logic programs, queries are executed against the program clauses, with the intent of unifying or (simply) verifying equality, on every term (variable or atom) with a possible match. The execution of a query over a program translates into a procedure of query *symbols* manipulation until a refutation is found. The refutation procedure used by Prolog was first mentioned by Kowalski [3] and later on named by Kowalski and Van Emden as Selective Linear Definite resolution (SLD resolution) [4]. Furthermore, a consolidation of the work was presented by Robinson [16], where a variant of the general refutation procedure was only used on definite clauses. A brief demonstration of the SLD resolution procedure is presented next.

Let us consider a query (*goal*), as a conjunction of subgoals, of the form

$$:- A_1, ..., A_{n-1}, A_n.$$

which we want to match against our program. First, and according to a $\text{select}_{literal}$ rule, a subgoal is selected for the initial unification with the program clauses.

Supposing that the subgoal chosen was $A_i$, the second step is to search the program for a clause that matches $A_i$. If program contains clauses in such conditions, the procedure continues by selecting the clause that will unify with $A_i$, according to a $\text{select}_{clause}$ rule.

Assuming that the selected clause to unify with $A_i$ has the form

$$A :- B_1, ..., B_m.$$

and that substitution $\theta$ represents the unification of both selected subgoal and clause, i.e., all the variables from the subgoal are bound with the variables from the selected clause. As a result our query became

$$:- (A_1, ..., A_{i-1}, B_1, ..., B_m, A_{i+1}, ..., A_n)\theta.$$

This procedure is repeated until a refutation is obtained. It is possible to obtain a successful SLD resolution when all subgoals are found to be true. The preformed substitutions will be a *(or the only)* possible answer to our query. On the contrary, if the procedure fails, implying an impossible unification between the query and the selected clause, the SLD resolution fails and no refutation of the query is possible. In this case, Prolog uses a *backtracking* mechanism to explore other possible unifications by, simply undoing the computations performed and selecting a different unification clause to our selected literal $A_i$. The specification of this procedure emphasizes the crucial role of the $\text{select}_{clause}$ and $\text{select}_{literal}$ rules. The application of different selection rules can lead to distinct solutions or otherwise solutions are presented

in a different order. Therefore, the specification of the selection rules is needed for real implementations. In the next section we describe Prolog's approach.

### 2.1.1 Prolog

Noticeable, Prolog is the most famous and used logic programming language. In 1972, Allain Colmerauer and Philippe Roussel began to develop a software tool to implement a man-machine system that would use natural language to communicate. The name Prolog was chosen as an abbreviation for "PROgrammation en LOGique" as a result of, language processing and automated theorem-proving mixing [17].

From Robinson's breakthrough presented in the *Resolution Principle* [16], Colmerauer and co-workers proceed their work by defining the semantics and the procedural method used by Prolog. In 1973, the demonstration of resolution and unification in Horn clauses [4] open new pathways to the definition of the fixed point semantics of Horn clause programming thus providing the necessary basis to prove that Prolog could be read, both procedurally and logically.

Being Prolog procedural semantics based on SLD resolution, the definition of the $\text{select}_{clause}$ and $\text{select}_{literal}$ rules was therefore necessary in order to possibilitate its implementation. In Prolog, the $\text{select}_{clause}$ rule follows the clauses order defined in the program code and the $\text{select}_{literal}$ rule chooses the leftmost subgoal in the query. In fact the first version of Prolog was a kind of automated deductive system, allowing development of a communication system in french. Additionally two other applications were also possible, such as a symbolic computation system and a general problem-solving system called Sugiton. The Second version of Prolog was more oriented towards actual programming language with the creation of the syntax, basic primitives and also the interpreter's computing method. The growth of Prolog as a programming language was aided by David H. D. Warren with his implementation of the first Prolog compiler in 1977 [18]. This development increased Prolog popularity offering the possibility of its syntax (*de facto* Prolog) to became a standard. In 1983, a new abstract machine was presented [1], able to execute compiled Prolog code, the Warren's Abstract Machine (WAM). Nowadays, the WAM is the most popular and efficient method of implementing Prolog and is actually the base of almost all Prolog systems.

Logic programming has indeed become an important core of computer science when Japan announced the Fifth Generation Project, with the intent to create a new Era for computer hardware based on artificial intelligence. As a result, many different Prolog models were created and literature for different levels of knowledge and audiences are now available [19, 20, 21]. Furthermore, the advances achieved in the implementations of Prolog and its compilation technology, brought the possibility to compare against imperative programming languages such as C [2]. Also, the inherent parallelism that seems to be

available in the logic programming paradigm became one of the ruling areas of interest giving Prolog the major importance and consideration in the current days.

### 2.1.2 The Warren's Abstract Machine

Some of Prolog's success is in part due to the accomplishments obtain by David H. D. Warren and his work on the efficient implementation of the WAM compiler to Prolog [1]. In fact, most of the logic programming systems still rely on the achievements of WAM's technology.

In a nutshell, the WAM consists basically, of a stack-based memory architecture allied to an instruction set, with simple data structures. At any time, the computation state can be obtained from WAM's data structures, data areas and registers. Figure 2.1 illustrates the composition of WAM's data structures and respective organisation.

The WAM's execution stack structure is composed by five different parts:

- **Push Down List (PDL)**: also known as unification stack, is used for the unification process;
- **Trail**: is organized as an array of addresses; used to store the address of (stack or heap) variables which must be *unbound* upon backtracking. Because it works like a stack we need to have a TR register that contains the reference to the top of the trail;
- **Stack**: also mentioned as the *local stack* is used to store the *environment frames* and the *choice point frames*:
  - **Environments**, store the information needed to continue execution after returning from a successful intermediate call. An environment is pushed into the stack whenever a clause contains more than one subgoal; an environment is popped out when the last clause's subgoal is executed. Each frame keeps the reference to the previous environment, thus giving the possibility to get the correct environment after the current one is popped out; and a set of cells, corresponding to the number of *permanent variables* in the body of the invoked clause. A permanent variable is a variable that appears in more than a subgoal in a clause's body. A register E is used to refer to the current active environment.
  - **Choice points**: store the information about the state of the computation for a procedure call, so that upon backtracking, the computation can be restored to the point when the choice occurred. In order to do so, all the data necessary to restore a computation is stored on a choice point. This includes the arguments of the current subgoal call; the reference to the continuation environment; a pointer to the next alternative clause; and pointers to the current values of the TR (*trail*)

and H (*heap*) registers. A choice point is pushed onto the stack whenever there is a point of choice and popped off when the last clause has no more alternatives. In order to access the sequence of choice points, the register B markes the current active choice point.

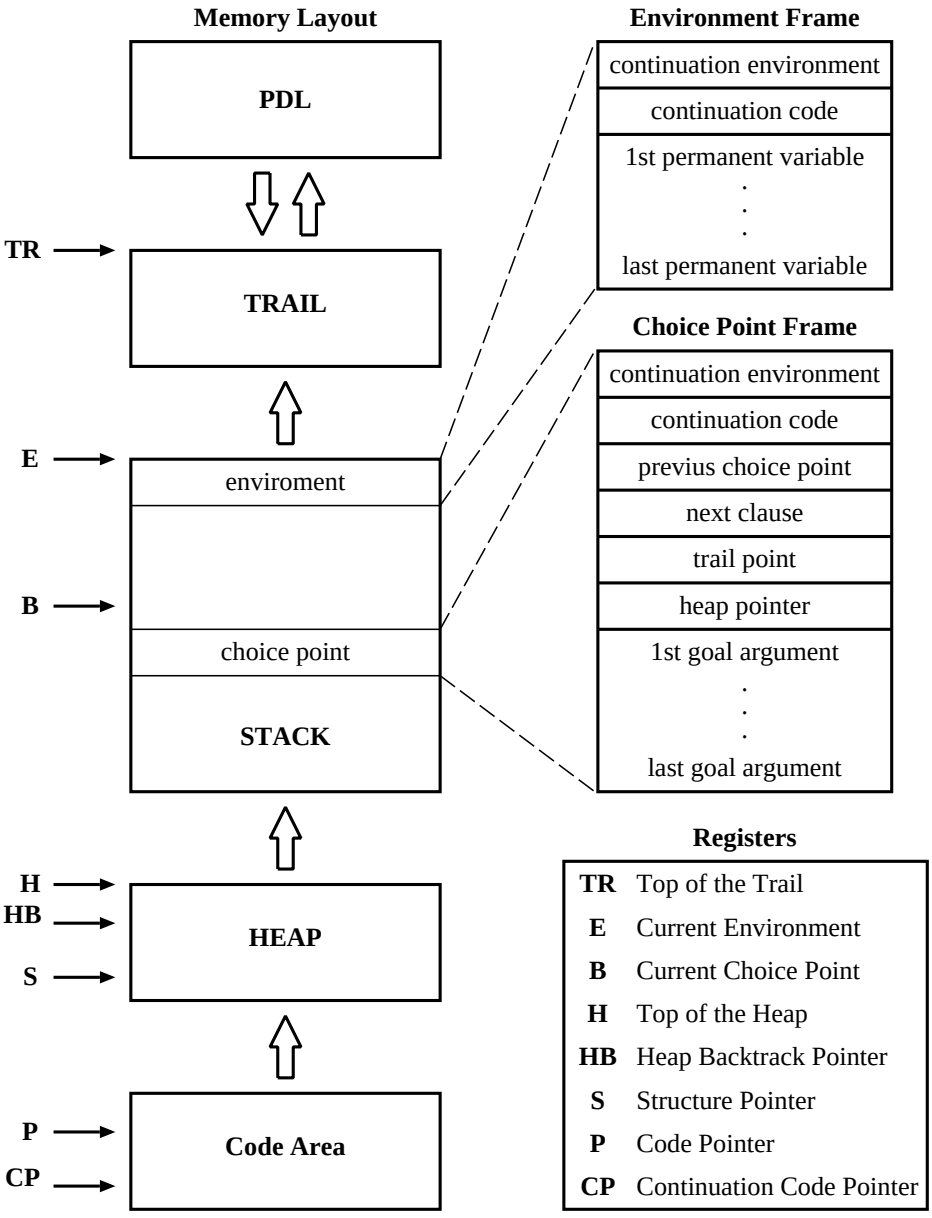



Figure 2.1: WAM memory layout, frames and registers description.

Some WAM implementations use two different stacks to store this structures, namely XSB [22] and SICStus Prolog [23]. As mentioned by H. Aït-Kaci [11], in such cases the two different stacks are the OR-stack (for the choice points) and the AND-stack (for

the environments).

- **Heap**: also referred as the *global stack*, is an array of data cells that is used to store the internal representation of Prolog terms, such as variables, atoms, structures or list terms. The register H contains the reference to the top of the heap.

- **Code Area**: an addressable array of data cells, consisting of op-codes followed by operands used to store WAM instructions for the (already) compiled program code.

Other important features of the WAM are also shown in Fig. 2.1, such as the register HB, that is used to contain the value of H, when a choice point is about to be created. All bindings done over variables after creating a choice point are considered *conditional bindings* meaning that they should be stored in the trail and therefore the value stored in HB is used to make such decision in the proper way. Another register is S which is used to help in the unification process by making reference to the point of the compound term where the unification process is in. Other referenced register is P which is set to maintain the address of the next instruction to execute in the *Code Area* (program counter). Finally, the register CP is used to reference (in the code area) to the location of the next instruction in the goal sequence, after successful return of a call.

The WAM structure and its components are handled by a simple set of instructions composed by:

- **Choice point instructions**, responsible for all interactions with choice points such as instructions to allocate/remove choice points and to recover the computation state using the information stored in choice points;

- **Control instructions** which interact with environments (allocate/remove) and also manage the call/return sequence of subgoals;

- **Unification instructions**, responsible for the implementation of specific versions of the unification algorithm according to the position and type of the arguments;

- **Indexing instructions**, used to accelerate the process of selecting the clauses that unify with a given subgoal call. The indexing procedure uses the first argument of a call, to jump to specialized code that is responsible to select only the unifying clauses.

Although the WAM appears as a simple system with a few groups of instructions it is indeed a very elaborated machine capable of executing all the complex mechanisms of Prolog. A complete and detailed specification of the WAM can be found in [11].

## 2.2 Tabling

Logic programming languages, like Prolog, use SLD resolution and Horn clauses for execution basis but, despite their power and declarativeness, they can suffer from some limitations. Those restrictions are the inability to deal with infinite loops and redundant sub-computations. This compromise the usage of Prolog and similar programming languages on important applications, such as Deductive Databases. Much work have been made to overcome those limitations by implementing strategies that remember sub-computation and its results, therefore avoiding re-computations and at the same time reusing the already stored answers. These techniques are known by several names like *memoizing, tabling* or *tabulation* [24].

Tabling [6] became a renowned technique thanks to the leading work in the XSB-Prolog system and, in particular in the SLG-WAM engine [7]. As a result several implementations of tabling mechanism were developed, having particular differences, namely in the execution rules, in the data-structures used to implement tabling and also in the underlying changes to the Prolog's engine. Examples of those implementations are available in systems such as YAP Prolog [25], B-Prolog [26] or ALS-Prolog [27]. The Tabling concept provides the basis to a bottom-up evaluation approach that, together with its well-know advantages, enables the combination with top-down evaluation, thus joining the better of both strategies.

### 2.2.1 Tabled Evaluation

The basic idea behind a tabled evaluation is, in fact, quite straightforward. The mechanism basically consists in storing all the different subgoal calls and new answers founded when evaluating a program in a proper data space called the *table space*. The subgoal calls stored in this table space are then used to verify if a subgoal is being called for the first time or, on the other hand, if it is a recall. Whenever such a repeated subgoal call occurs, the answers for that subgoal (stored in the table space) are used instead of re-evaluating the subgoal against the program clauses. Next we present a simple demonstration of a table evaluation that emphasizes the tabling technique advantages. Consider the Prolog program shown in the top of Fig. 2.2 representing a small directed graph. The predicate `arc/2` represents the direct connection between two different points and the `path/2` predicate represents the possibility of an indirect connection. Consider now the query goal `path(1,Z)`. An direct application of SLD evaluation to solve the given query leads to an infinite SLD tree, as shown in the bottom of Fig. 2.2, due to the positive loop induced by the selection of the leftmost literal rule. On the other hand, when resorting to tabling, the infinite search tree resulting from the positive loop will not occur, and termination is ensured. The scheme presented in Fig. 2.3 shows the evaluation sequence when using tabling (solving the same query in the same program).

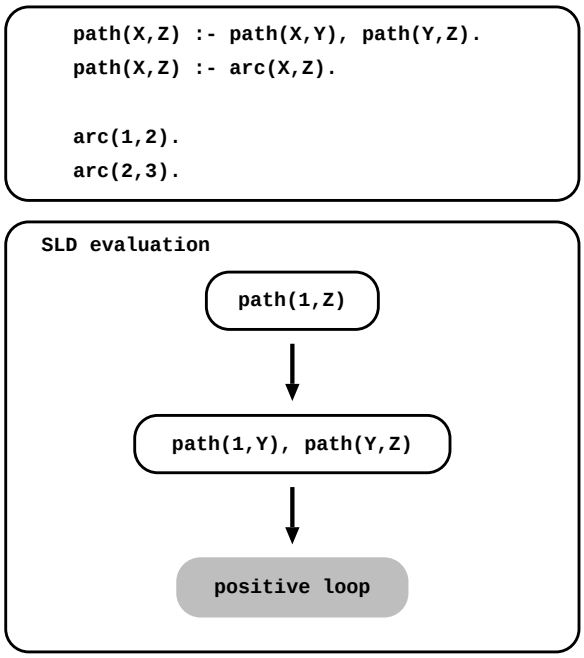



Figure 2.2: An infinite SLD evaluation.

Figure 2.3 shows a small change on the Prolog code compared to the one presented in Fig. 2.2, namely the declaration `:- table path/2`, indicating that the tabling procedure should be applied to all the subgoal calls to `path/2`. Those subgoal calls can be seen in the top right of Fig. 2.3 on the representation of the table space at the end of program's evaluation. The bottom of the figure shows the resulting trees created whenever a tabled subgoal call is made for the first time (nodes 0, 5 and 11). The answers resulting from the evaluation of new trees are store in the respective table entry, so those answers can be used when variant calls (such as the nodes 1, 6 and 12) occur. When a variant call consumes all the answers stored in the table space, or in case of their absence, the evaluation is suspended. In the meantime, if new answers arise the suspended variant calls are resumed to properly consume the new answers. In this way, the re-evaluation of variant calls is avoided.

During this process, the table space structure has a main role, not only because it is the core of the tabling implementation but also because it will be involved in the most of the tabled evaluations interactions. In fact, the performance of tabling depends on the implementation of the table space itself, being critical for the success of the tabling implementation. Therefore a well defined and efficient data structure is needed. Arguably, the most successful data structure for tabling is *tries* [28, 8].

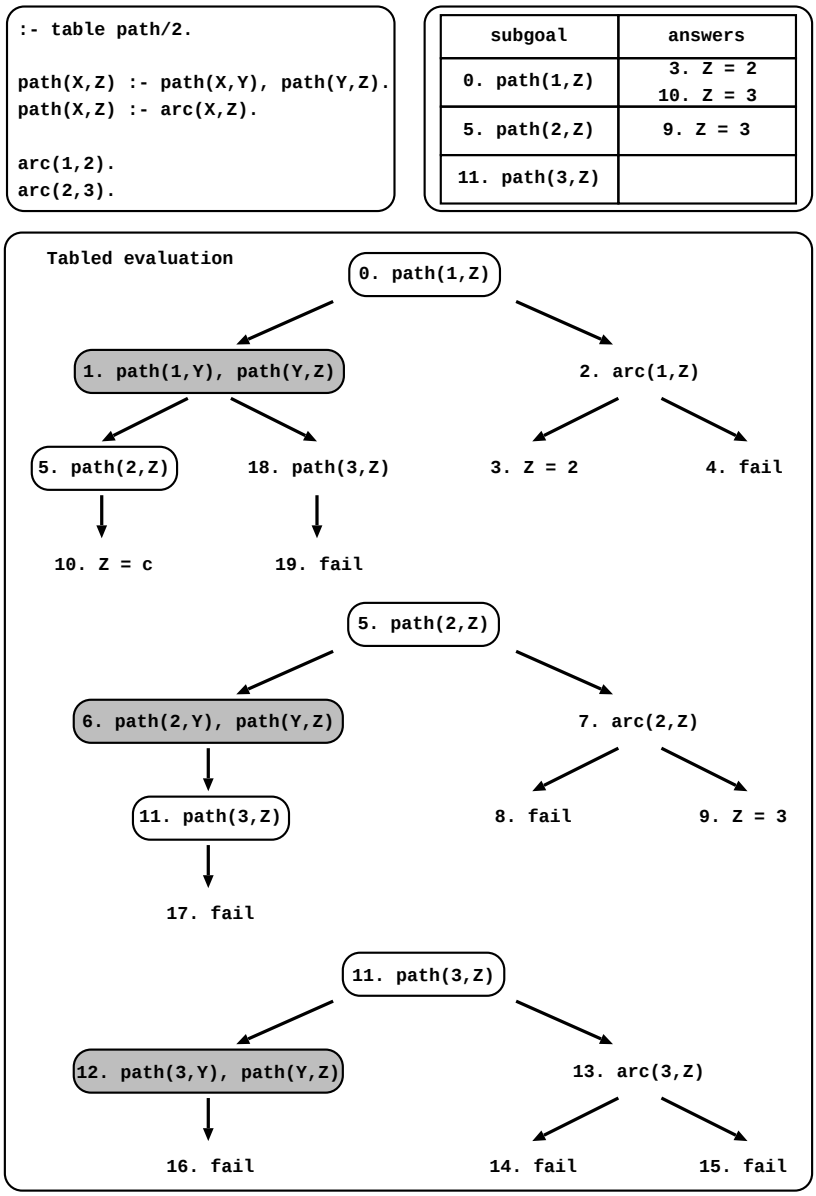
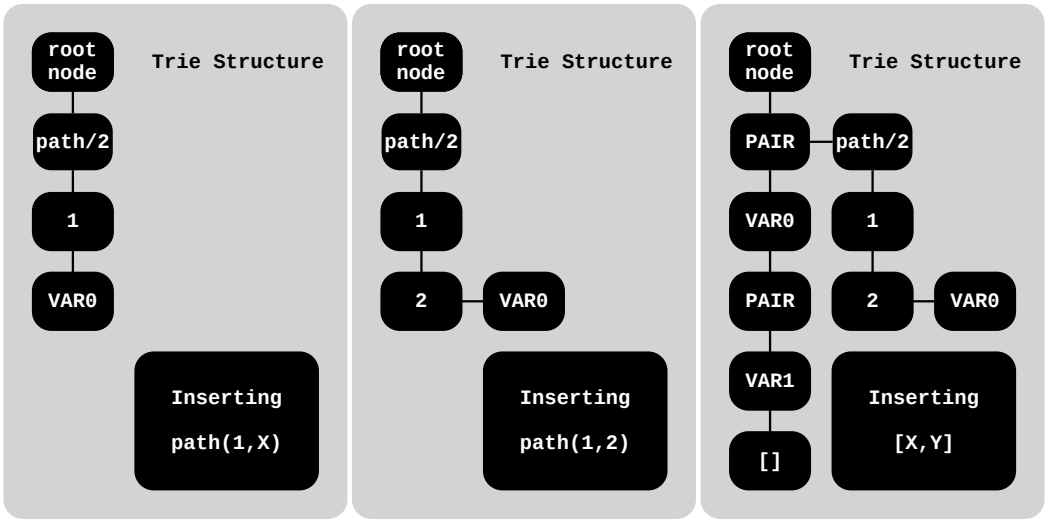



Figure 2.3: A finite tabled evaluation example.

### 2.2.2 Tries

The table space can be accessed in many different ways. A well defined and efficient data structure is supposed to give response to interactions such as; **(i)** finding a subgoal in a table and, if not present, insert it; **(ii)** verify whether a founded answer is already stored in a table and, if not, insert it; and **(iii)** loading answers from tables to variant calls. The YapTab engine uses tries as proposed by I.V. Ramakrishnan *et al.* [28, 8] which is considered to be a very effective way to implement the table space. A trie is a structure like a tree, where every

different path connecting different trie nodes (the unit data for tries) corresponds to a single term representation, that can be seen as a tokenized form of terms, as illustrated in Fig. 2.4.




(a) (b) (c)

Figure 2.4: Representing terms in a trie.

Figure 2.4a shows the representation of term `path(1,X)` in a trie as a sequence of three tokens: the token `path/2` to represent the functor's name and arity, the token `1` to represent the atom with the same name, and finally the token `VAR0` to represent the variable `X` present in the term. Variables are represented using the formalism proposed by Bachmair *et al.* [29] where each variable in a term is represented as a distinct constant $VAR_i$. If another term is inserted in the same trie having a common prefix to the already inserted one, tries have the property to not represent the equal part of the term. As shown in Fig. 2.4b, when inserting the term `path(1,2)` with the token representation $<$ `path/2, 1, 2` $>$ it only differs in token `2`, from the previous term, thus adding it to the trie, corresponds to insert a trie node for token `2` as a sibling of the trie node where the difference between both terms first occurs. Finally, if a term differs in the very beginning of its tokenized form, a new entry is added to the top of the trie as shown in Fig. 2.4c with the insertion of the term `[X,Y]` corresponding to the tokenized form $<$ `Pair, X, Pair, Y, []` $>$[1]. With this example, it can be easily seen the compactness propriety of term representation in tries.

To obtain the best performance from tries usage, the YapTab system applies two levels of tries in the implementation of the table space, a top level for the subgoal calls and a second level for computed answers. For every tabled predicate is created beforehand a subgoal trie where the root node marks the entry point for insertion of the corresponding subgoal calls.

[1] Lists representation will be covered in more detail in a later chapter.

At this level each path in the subgoal trie represents a distinct subgoal call where the leaf node acts as a connection with second level of tries (the *answer trie*) through the use of an auxiliary data structure, called subgoal frame. In the answer trie, all the computed answers for the respective subgoal are stored, once again every path corresponding to a unique answer.

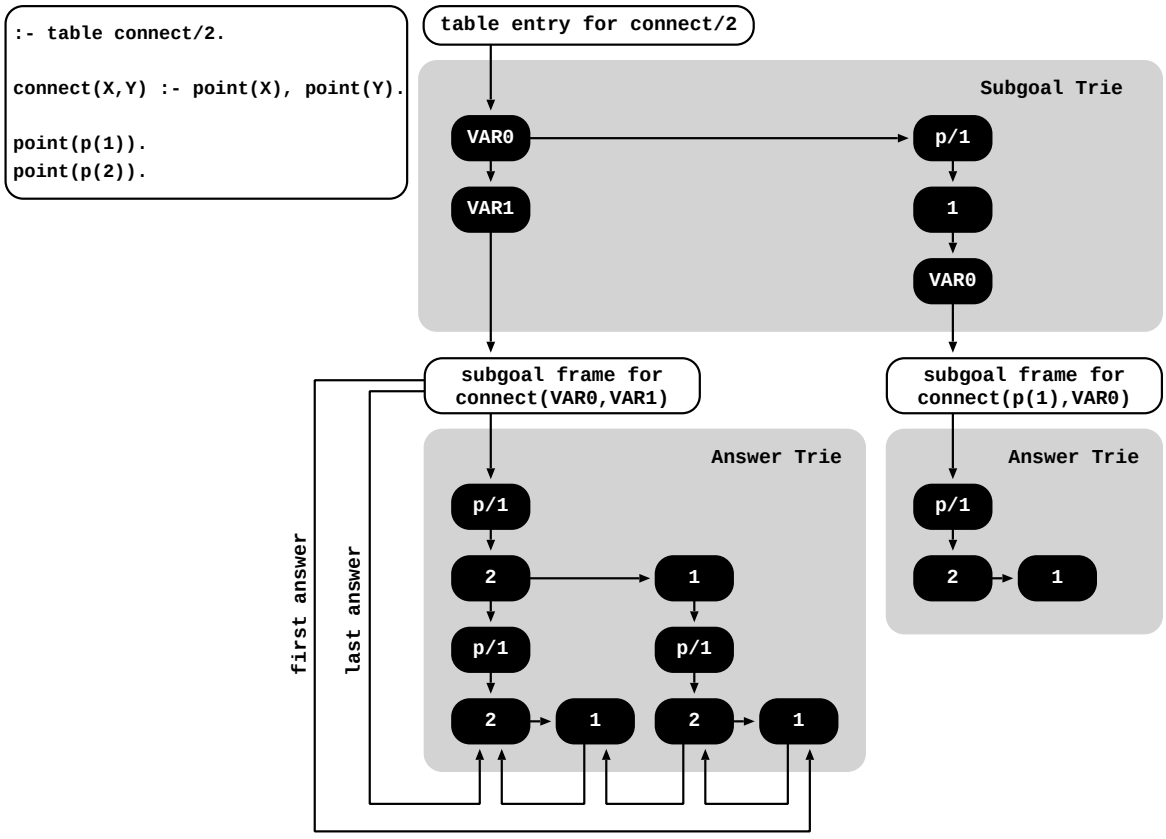



Figure 2.5: YapTab table space organization.

The previous description can be observed in more detail in the YapTab table space structure presented in Fig. 2.5. In this example we can see two different subgoal calls for a predicate `connect/2`. The subgoal call `connect(p(1),X)`, inserts nodes to represent the term `p(1)` and the variable `X` (`VAR0`), and also adds the respective subgoal frame. The subgoal call `connect(X,Y)` as differs in the first element of the call, leads to inserting the nodes for `VAR0` and `VAR1`, representing respectively the variables `X` and `Y`, and once more a subgoal frame is also created. Regarding the answer tries, for the subgoal call `connect(p(1),X)`, the answer trie has two different answers, corresponding to the possible values that can be instantiated to `X`, `p(1)` and `p(2)`. In this case three nodes are inserted to represent the two solutions: a common node to represent `p/1` and two more to represent the constants `1` and `2`. On the other hand, for the subgoal call `connect(X,Y)`, the answer trie represents all the answers obtained by combining all the values that can be instantiated to `X` and `Y`.

Another feature, from YapTab's table space organization, that is illustrated in Fig. 2.5 is the connection between the leaf nodes existing in an answer trie. This linked list is used to maintain a chronological order of the insertion of answers, and the respective subgoal frame has a pointer to the first and last solutions inserted. This feature is of a major importance because when a variant call is suspended, it only needs to keep a reference to the last consumed answer, as afterwards, when the computation is resumed, answer consumption can start from that reference if there are new solutions to consume.

### 2.2.3 Compiled Code on Tries

On completion of an answer trie, from a given subgoal trie, an optimization exists that avoids answer recovery with a bottom-up strategy, i.e., with terms being loaded starting from the leaf nodes. Instead, the answer tries are dynamically compiled into WAM-like instructions from answer trie nodes, enabling a top-down traverse of the trie to consume answers. These compiled instructions are called *trie instructions* and the restructured tries are called *compiled tries* [8]. Compiled tries are shared during execution of the trie instructions, therefore when backtracking from a certain term, the procedure continues by loading the term sibling node, keeping the remaining structure of the term. In this manner, each node of the trie is traversed only once, benefiting of the compactness of term representation in tries. In Fig. 2.6 we have an example of a compiled trie for the subgoal call `connect(X,Y)` presented in Fig. 2.5.

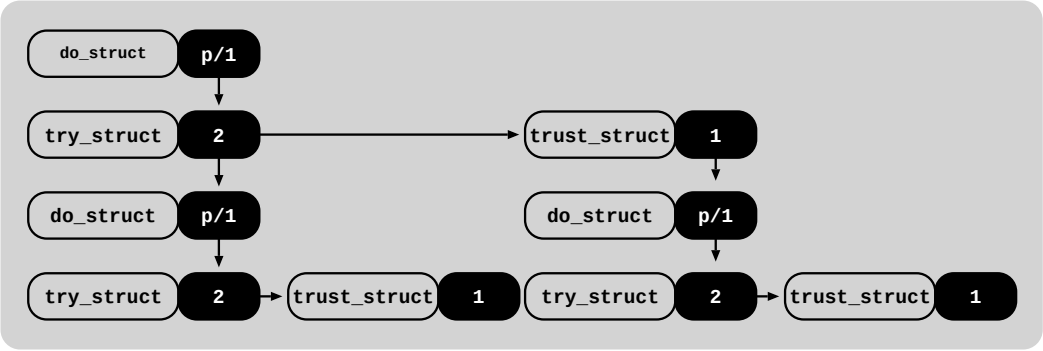



Figure 2.6: Compiled trie for the subgoal call connect(X,Y) presented in Fig. 2.5.

In compiled tries each node is combined with an instruction, the selection of the instruction is influenced by the term type represented in the trie node and by the position of the node in the respective list of possible sibling nodes. Therefore trie instructions can be grouped into four different types, since each trie node can appear as the first, intermediate, last or the only sibling of a sequence. Namely, first position sibling nodes are compiled using `try_?` instructions, intermediate nodes using `retry_?` instructions, last nodes using `trust_?`

instructions and, if a node is the only sibling, using `do_?` instructions. Each instruction also refers the term type in the trie node, for example with atom terms the possible instructions are `try_atom, retry_atom, trust_atom` or `do_atom`. At the engine level, compiled trie instructions act similarly to the generic *try/retry/trust* WAM instructions but, in this case, they are responsible for interacting with choice points to correctly traverse top-down an answer trie, in such way that, in case of failure, the procedure continues to the next sibling node. The *do* instruction denotes no choice and thus no choice point allocation is preformed.

# Chapter 3

# List Terms Representation

In this section, we first introduce YapTab's design for the representation of list terms, and then we present our new and alternative design for list term representation [30] which the main goal is to optimize YapTab's memory usage in order to reduce possible drawbacks of the standard table mechanisms. In what follows, we will refer to the original design as *standard lists* and to the new design as *compact lists*. We start by briefly introducing how standard lists are represented in YapTab and then we discuss in more detail the new design for the representation of compact lists.

## 3.1 Standard Lists

YapTab follows the seminal WAM representation of list terms [11]. In YapTab, list terms are recursive data structures implemented as functors of two elements, named *pairs*, where the first pair element, the *head* of the list, represents a list element and the second pair element, the *tail* of the list, represents the list continuation term or the end of the list. In YapTab, the end of the list is represented by the empty list atom `[]`. At the engine level, a pair is implemented as a pointer to two contiguous cells, the first cell representing the head of the list and the second the tail of the list. In YapTab, the tail of a list (or the second element of a pair) can be any term (and not only another pair or the empty list atom). Figure 3.1(a) illustrates YapTab's WAM representation for list terms in more detail.

Alternatively to the standard notation for list terms, we can use the pair notation `[H|T]`, where `H` denotes the head of the list and T denotes its tail. For example, the list term `[1,2,3]` in Fig. 3.1 can be alternatively denoted as `[1|[2,3]]`, `[1|[2|[3]]]` or `[1|[2|[3|[]]]]`. The pair notation is also useful when the tail of a list is neither a continuation list nor the empty list. This list term's type representation can be seen for example in the list `[1,2|3]` shown in Fig. 3.1(a) by its corresponding WAM representation. In what follows, we will refer to these

lists as *term-ending lists* and to the most common lists ending with the empty list atom as *empty-ending lists*.

Regarding the trie representation of lists, the original YapTab design, as most tabling engines, including XSB Prolog, tries to mimic the corresponding WAM representation. This is done by making a direct correspondence between each pair pointer at the engine level and a trie node labelled with the special token `PAIR`. For example, the tokenized form of the list term `[1,2,3]` is the sequence of seven tokens <`PAIR`,`1`,`PAIR`,`2`,`PAIR`,`3`,`[]`>. Figure 3.1(b) shows in more detail YapTab's original trie design for the list terms represented in Fig. 3.1(a).

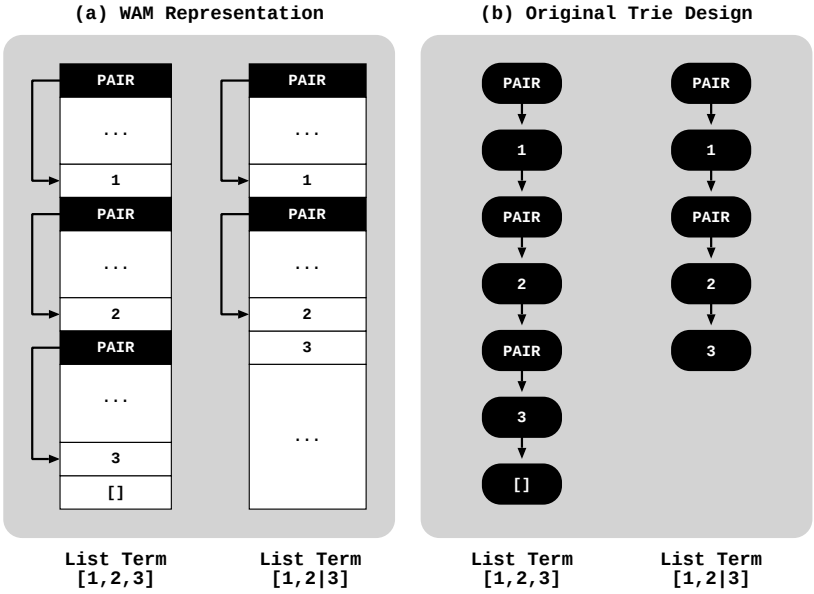



Figure 3.1: YapTab's WAM representation and original trie design for standard lists.

## 3.2 Compact Lists

In this section, we introduce the new design for the representation of list terms. The discussion we present next tries to follow the different approaches that we have considered until reaching our current final design. The key idea common to all these approaches is to avoid the recursive nature of the WAM representation of list terms and have a more compact representation where the unnecessary intermediate `PAIR` tokens are removed, therefore reducing the system memory when storing lists.

Figure 3.2 illustrates how compact lists are represented in tries using our initial approach. Comparing with Fig. 3.1, in this approach, all intermediate `PAIR` tokens are removed and a compact list is simply represented by its term elements surrounded by a begin and a end

list mark, respectively, the `BLIST` and `ELIST` tokens. Figure 3.2(a) shows the tokenized form of the empty-ending list `[1,2,3]` which, with this design, is the sequence of six tokens <`BLIST`,`1`,`2`,`3`,`[]`,`ELIST`>, and the tokenized form of the term-ending list `[1,2|3]` which, with this design, is the sequence of five tokens <`BLIST`,`1`,`2`,`3`,`ELIST`>. This approach clearly outperforms the standard lists representation when representing individual lists, with a unique exception happening when constructing the basic cases of list terms of size one to three. When representing individual list terms with more than three elements it requires about half the nodes required for standard lists. For an empty-ending list of $S$ elements, standard lists requires $2S + 1$ trie nodes and compact lists requires $S + 3$ nodes. Regarding term-ending lists of $S$ elements, standard lists representation requires $2S - 1$ trie nodes, and yet when using compact lists it requires $S + 2$ nodes.

Next, in Fig. 3.2(b) we try to illustrate how this approach behaves when we represent more than a list in the same trie. It presents three different situations: the first situation, shows two lists with the first element different and it illustrates a kind of worst case scenario when representing list terms in a trie; the second and third situations show, respectively, two empty-ending and two term-ending lists with only the last element different, that can be seen as a kind of best case scenario when representing list terms in a trie, which means that only the last element of the second list representation is added to the trie.

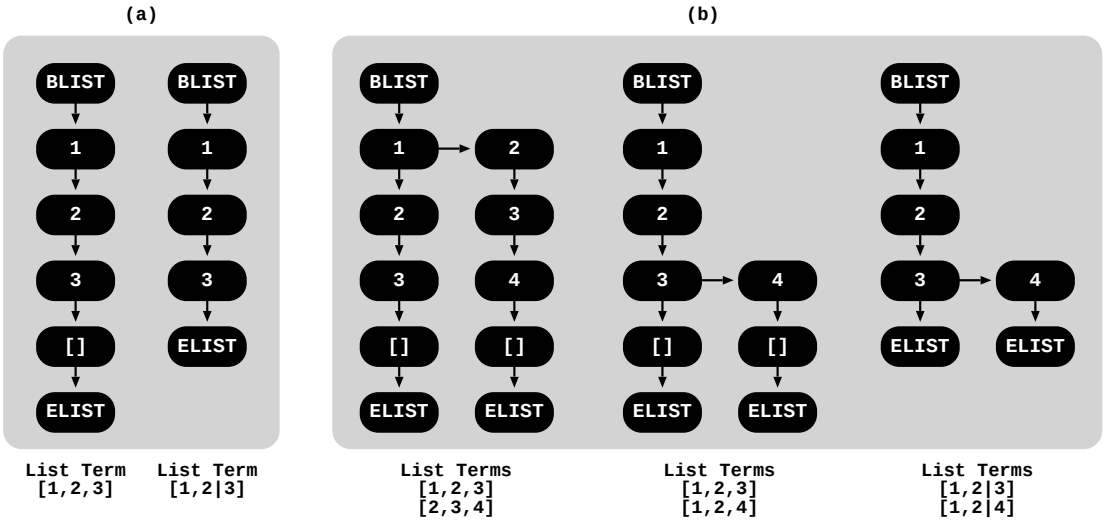



Figure 3.2: Trie design for compact lists: initial approach.

Now consider that we generalize these situations and represent in the same trie $N$ lists of $S$ elements each. For the first situation (when lists differ in the first element) our first approach is always better than standard lists, but this may not be the case when it regards the second and third situations. For the second situation (empty-ending lists with last element different),

standard lists representation requires $2N+2S-1$ trie nodes and compact lists requires $3N+S$ nodes and thus, if $N > S-1$, i.e., if the number of distinct lists are greater than the size of the list represented, then standard lists representation has better results, requiring less nodes to represent lists in such conditions. Regarding the third situation (term-ending lists with last element different), standard lists requires $N+2S-2$ trie nodes to represent lists in these conditions and compact lists requires $2N+S$ nodes, and once again if $N > S-2$, then standard lists representation spend less nodes when representing list terms.

When analysing the representation of compact lists in this approach, the main problem is the introduction of the extra token `ELIST` in the end of each different list, the cost of this extra token is more evident when representing lists with the last element different, because instead of adding only one node (the different one), for each different list, we add two nodes.

To avoid this problem, we have redesigned our compact lists representation in such a way that the `ELIST` token appears only once for lists with the last element different. Figure 3.3 illustrates our second approach for the compact lists representation, using the same lists presented previously in Fig. 3.2.

In this second approach, a compact list still contains the begin and end list tokens, `BLIST` and `ELIST`, but now the `ELIST` token plays the same role of the last `PAIR` token in standard lists, i.e., it marks the last pair of terms in the list. Figure 3.3(a) shows the new compact list tokenized form obtained when using this change. The empty-ending list `[1,2,3]` is now represented as <`BLIST`,`1`,`2`,`ELIST`,`3`,`[]` > and the new tokenized form of the term-ending list `[1,2|3]` is now represented by <`BLIST`,`1`,`ELIST`,`2`,`3`>. To verify how this second approach behaves when we represent more than a list in the same trie, in Fig. 3.3(b) we illustrate the same three situations of Fig. 3.2(b). For the first situation (lists with the first element different), the second approach is identical to the initial approach. This is straightforward since the changes made simply move the `ELIST` token from the end of the list, therefore the repetition of the `ELIST` token still occurs. For the second and third situations, the second approach is not only better than the initial approach, since it avoids the repetition of the `ELIST` token in the end of list representation, but also better than the standard lists representation, reducing the exceptions to the base cases of list terms of sizes 1 and 2.

Consider again the generalization to represent in the same trie $N$ lists of $S$ elements each. Since no changes occurred in the first situation, this second approach has the same results as the first approach. On the other hand, for the second situation (empty-ending lists with last element different), compact lists now requires $2N+S+1$ trie nodes (the initial approach for compact lists required $3N+S$ nodes and standard lists required $2N+2S-1$ nodes) and for the third situation (term-ending lists with last element different), compact lists now requires $N+S+1$ trie nodes (the initial approach for compact lists required $2N+S$ nodes and standard lists required $N+2S-2$ nodes). Despite these better results, this second approach

still contains some drawbacks that can be improved.

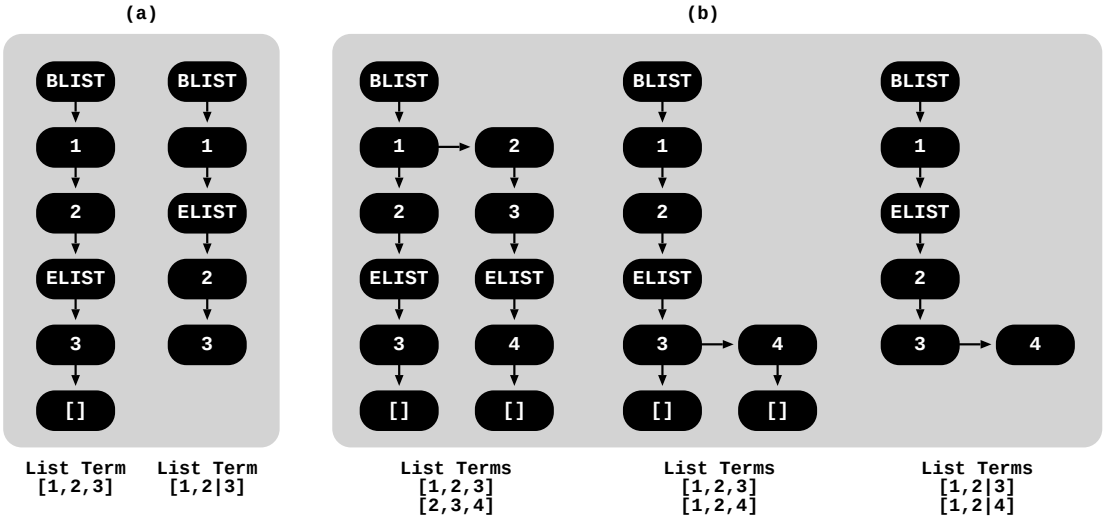



Figure 3.3: Trie design for compact lists: second approach.

Figure 3.4 illustrates our final approach for the representation of compact lists. In this final approach, we have redesigned our previous approach in such a way that the empty list token `[]` was avoided in the representation of empty-ending lists. Note that, in our previous approaches, the empty list token is what allows us to distinguish between empty-ending lists and term-ending lists. So, in order to maintain this distinction, we did not simply removed the empty list token from the representation of compact lists. To provide the needed distinction between lists, we added a different end list token, `EPAIR`, for term-ending lists, maintaining the `ELIST` token to represent empty-ending lists. Furthermore, we changed the behavior of the token representing the end of a list, instead of marking the last two elements of a list element, tokens `ELIST` and `EPAIR` are used to mark the last element in an empty-ending list and in an term-ending list, respectively. Figure 3.4(a) shows the new tokenized form of the empty-ending list `[1,2,3]`, which is now represented as <`BLIST`,`1`,`2`,`ELIST`,`3`>, and the new tokenized form of the term-ending list `[1,2|3]`, which is now represented as <`BLIST`,`1`,`2`,`EPAIR`,`3`>. Figure 3.4(b) shows how this final approach behaves when we represent more than a list in the same trie, using the same three previous situations for representing lists (different in the first element or different in the last element). For the three examples, this final approach clearly outperforms all the other representations for standard lists and previous approaches of compact lists. Regarding lists with the first element different (first situation), our final approach requires $N + NS + 1$ trie nodes for both empty-ending and term-ending lists, thus reducing the cost for the empty-ending lists representation, since the modifications were mainly made over the empty list token.

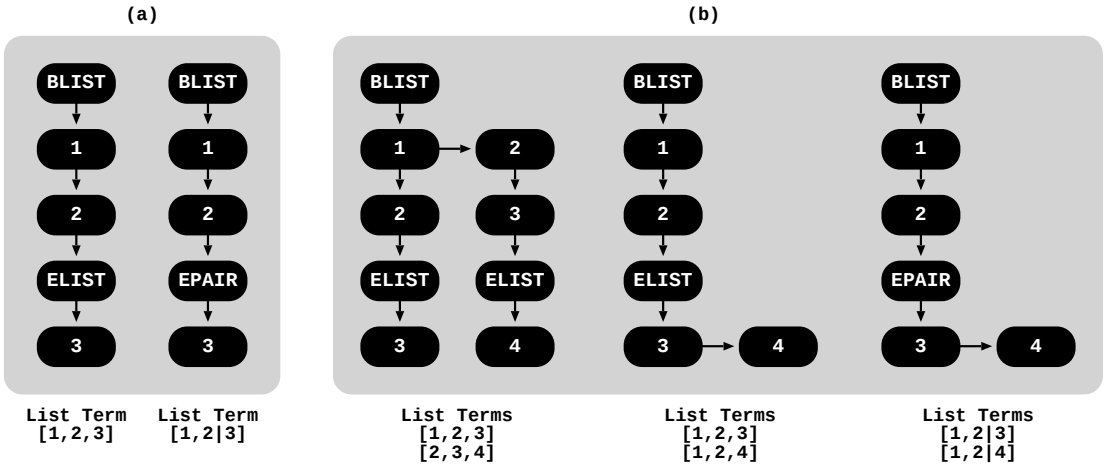



Figure 3.4: Trie design for compact lists: final approach.

Toward lists with the last element different (second and third situations), it requires $N + S + 1$ trie nodes for both empty-ending and term-ending lists, once again this change only takes effect on empty-ending lists. Table 3.1 summarizes the comparison between all the approaches regarding the number of trie nodes required to represent in the same trie $N$ list terms of $S$ elements each.

| *List Terms* | *Standard Lists* | *Compact Lists* | | |
|---|---|---|---|---|
| | | *Initial* | *Second* | *Final* |
| **First element different** | | | | |
| $N$ $[E_1, ..., E_{S-1}, E_S]$ | $2N + 2NS + 1$ | $2N + NS + 1$ | $2N + NS + 1$ | $N + NS + 1$ |
| $N$ $[E_1, ..., E_{S-1} \mid E_S]$ | $2NS + 1$ | $N + NS + 1$ | $N + NS + 1$ | $N + NS + 1$ |
| **Last element different** | | | | |
| $N$ $[E_1, ..., E_{S-1}, E_S]$ | $2N + 2S - 1$ | $3N + S$ | $2N + S + 1$ | $N + S + 1$ |
| $N$ $[E_1, ..., E_{S-1} \mid E_S]$ | $N + 2S - 2$ | $2N + S$ | $N + S + 1$ | $N + S + 1$ |

Table 3.1: Number of trie nodes to represent in the same trie N list terms of S elements each, using the standard lists representation and the three compact lists approaches.

## 3.3 Compiled Tries for Compact Lists

In this section, we discuss the implications of the new design in the completed table optimization and describe how we have extended YapTab to support compiled tries for compact lists. First we illustrate in Fig. 3.5(a) the compiled trie code for the standard list `[1,2,3]`. When using standard lists, each `PAIR` token is compiled using one of the

`try/retry/trust/do_list` trie instructions. At the engine level, these instructions create a new pair term in the heap stack to be bound to the term being constructed. In Fig. 3.5(b), we show the compiled trie code for the last compact lists approach. As mentioned, the initial step for compact list consisted in the removal of the `PAIR` tokens. Hence, we need to include the pair terms creation step in the trie instructions associated with the elements in the list, except for the last list element. To do that, we have extended the set of trie instructions for each term type with four new specialized trie instructions: `try_?_in_list`, `retry_?_in_list`, `trust_?_in_list` and `do_?_in_list`. As an example, for atom terms, the new set of trie instructions is: `try_atom_in_list`, `retry_atom_in_list`, `trust_atom_in_list` and `do_atom_in_list`. At the engine level, these instructions create a new pair term in the heap stack to be bound to the term being constructed and then they bind the head of the new pair to the sub-term corresponding to the `?_in_list` instruction at hand. Last list elements are treated as before and `ELIST` tokens are compiled using a new `?_ending_list` trie instruction. At the engine level, the `?_ending_list` instructions also create a new pair term in the heap stack to be bound to the term being constructed and, in order to denote the end of the list, they bind the tail of the new pair to the empty list atom `[]`. Finally, the `BLIST` and `EPAIR` tokens are compiled using `?_void` trie instructions. This type of instructions do nothing since the construction of the heap terms is done by the `?_in_list` instructions.

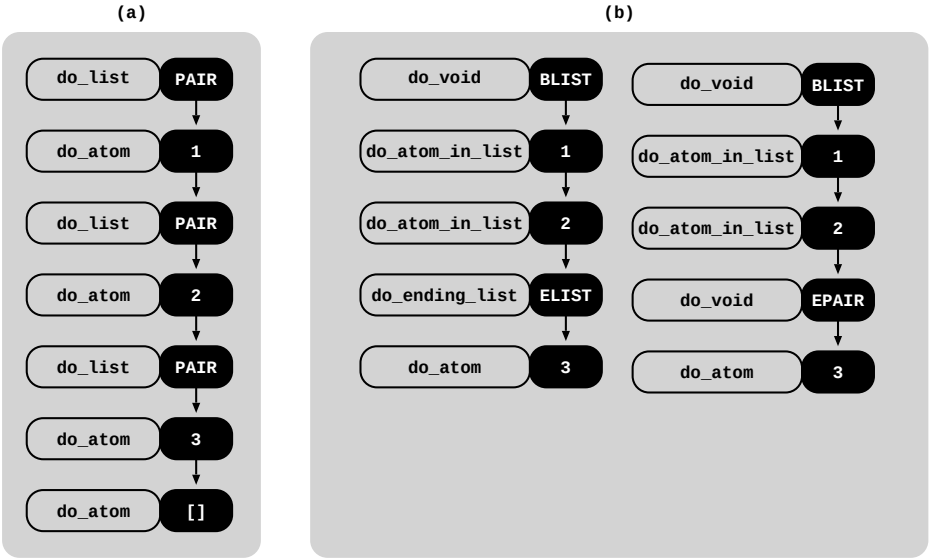



Figure 3.5: Comparison between the compiled trie code for standard and compact lists.

Note however that the trie nodes for the tokens `BLIST` and `EPAIR` cannot be avoided because they are necessary to distinguish between a term `t` and the list term whose first element is `t`, and to mark the beginning and the end of list terms when traversing the answer tries nodes bottom-up.

Next, we present in Fig. 3.6, two more examples showing how list terms including compound terms, the empty list term and sub-lists are compiled using the compact lists representation. In the left side of Fig. 3.6, we illustrate the tokenized form of the list term `[f(1,2),[],g(a)]` with the sequence of eight tokens <`BLIST`,`f/2`,`1`,`2`,`[]`,`ELIST`,`g/1`,`a`> and, on the right side of the figure, we illustrate the tokenized form of the list term `[1,[2,3],[]]` with the sequence of eight tokens <`BLIST`,`1`,`BLIST`,`2`,`ELIST`,`3`,`ELIST`,`[]`>. To see how the new trie instructions for compact lists are associated with the tokens representing list elements, we next present the previous tokenized forms, but with the tokens representing common list elements explicitly aggregated:

`[f(1, 2),[],g(a)]`: <`BLIST`,<`f/2`,`1`,`2`>,`[]`,`ELIST`,<`g/1`,`a`> >

`[1,[2, 3],[]]`: <`BLIST`,`1`,< `BLIST`,`2`,`ELIST`,`3`>,`ELIST`,`[]`>.

The tokens that correspond to first tokens in each list element, except for the last list element, are the ones that need to be compiled with the new `?_in_list` trie instructions (please see Fig. 3.6 for full details).

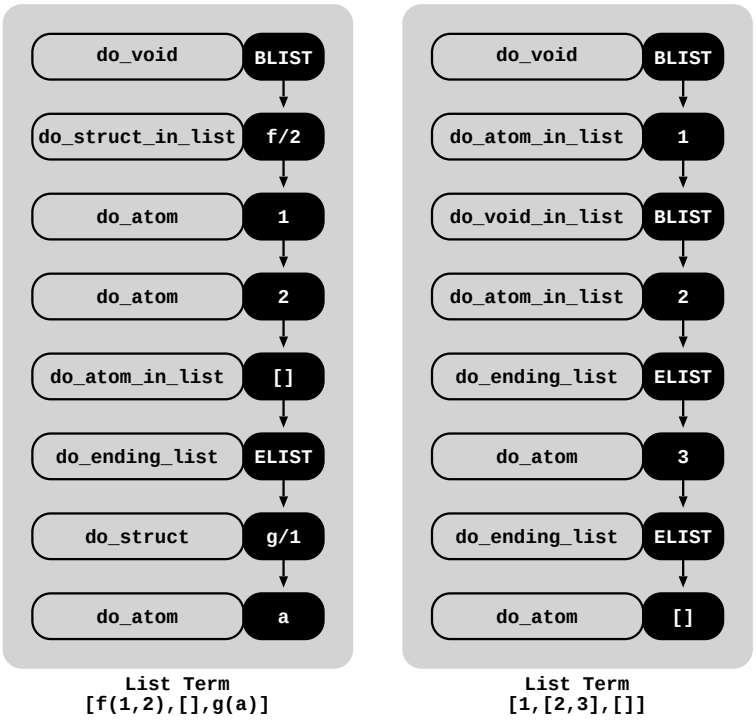



Figure 3.6: Compiled trie code for compact lists including compound terms and sub-lists.

In more detail, in list `[f(1,2),[],g(a)]`, the tokens to be compiled with the new `?_in_list` trie instructions are the tokens `f/2` and `[]`. Token `f/2` because it is the first token in the

aggregated representation <f/2,1,2> of the first list element and token [] because it is the single token representing the second list element. In the second example, list [1,[2,3],[]], as the second list element is itself a list, the same idea is applied not only to the tokens in the aggregated representation of the main list but also to the tokens in the aggregated representation, <BLIST,2,ELIST,3>, of the sub-list. Therefore, the tokens 1 (first element of the second element of the main list), BLIST (first token of the second element of the main list), 2 (first element of the sub-list) are compiled with the ?_in_list instruction.

# Chapter 4

# Global Trie

The tabling technique was developed to overcame particular limitations of Prolog. Nevertheless, when used to solve real world problems, tabling can show some drawbacks. One of the most common limitations of tabling, is the overload of system's memory. The Global Trie (GT) design stands as an alternative method to YapTab's standard table space representation. The GT table space design emerges with the intent to surpass those disadvantages, namely by storing terms in the same trie, thus preventing repeated representations of a term in different trie data structures. In this chapter, we describe the implementation of distinct GT's strategies.

## 4.1 Global Trie for Calls and Answers

As proposed by Costa and Rocha [31, 32], in the *Global Trie for Calls and Answer* (GT-CA) design, the main idea is to avoid term repetitions, which could take place in different trie data structures as shown in Fig. 4.1. Here, the representation of the terms `f(1)` and `f(2)` occurs several times each. The first approach to prevent these repetitions resorted to group all tabled subgoal calls and/or answers, by storing them in a common global trie, instead of being spread over several different tries. This conceptual change is achieved without removing the gains obtained by the use of tries. Therefore, the GT-CA data structure is still a tree structure, where each different path through the GT nodes corresponds to a subgoal call and/or answer. In spite of the new organization for the table space, the hierarchical structure of the table space still follows by the existence of a subgoal trie and an answer trie data structures (see Fig. 4.2). However, in this particular design, both are represented by a unique level of trie nodes that point to the corresponding terms in the GT-CA (see the `callN` nodes for the subgoal trie and the `answerN` nodes for the answer trie in Fig. 4.2). Henceforth, coexisting terms on calls and/or answers, are represented only once in GT-CA, thus avoiding

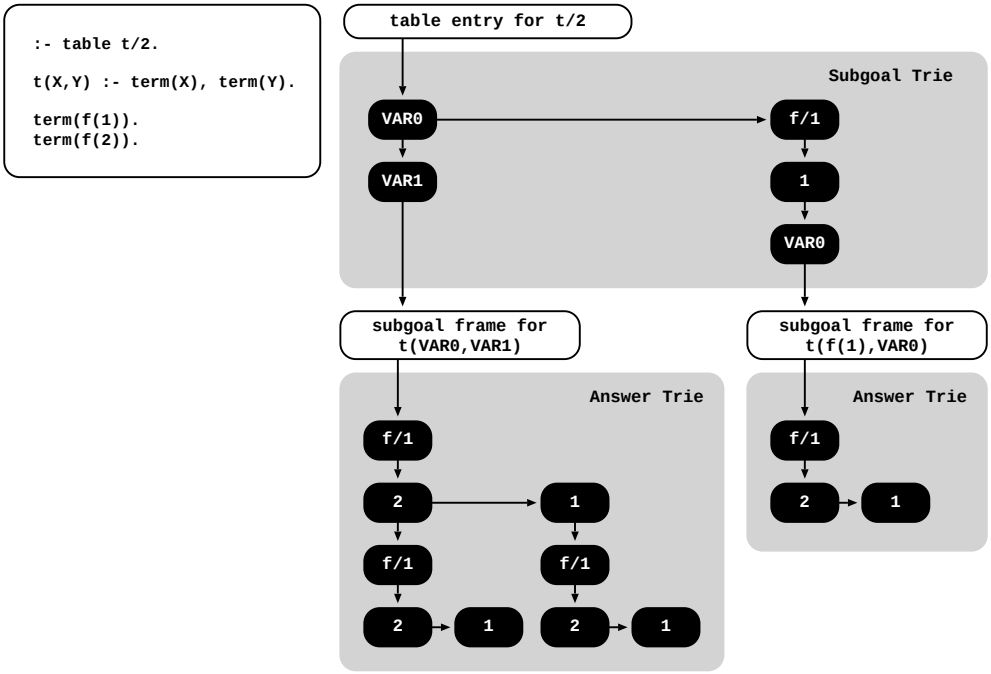



Figure 4.1: YapTab's standard table design.

repetition of terms once stored in the GT.

The role for the several tries is of simple assimilation. For the subgoal tries, each node now represents a different subgoal call. The node's token is the pointer to the node in the GT-CA corresponding to the path representation for the subgoal call, i.e., all argument terms represented in the original subgoal trie (Fig. 4.1) are now represented and inserted in the GT-CA. However, the organization used in the subgoal tries allows one to maintain the list of sibling nodes and the access to the corresponding subgoal frames unaltered.

In a similar way, for the answer tries, each node now represents a different answer for the respective subgoal. Instead of having the complete answer term represented in the answer tries, with this design the answer trie node's token is simply a pointer to the corresponding path in the GT-CA representation. Once again, the organization used in the answer tries to maintain the list of sibling nodes and to enable answer recovery in insertion order, remains unaltered. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing the corresponding GT-CA's nodes bottom-up.

Figure 4.2 uses the example from Fig. 4.1 to illustrate how the GT-CA design works. Initially, the subgoal trie and the GT-CA are empty. Then, the subgoal `t(f(1),X)` is called. When this occurs, three nodes are inserted in the GT-CA to represent the call: one represents the functor `f/1`, a second refers to the constant `1` and the last representing the variable `X`. Next, a node representing the path inserted in the GT-CA is stored in the subgoal trie (node

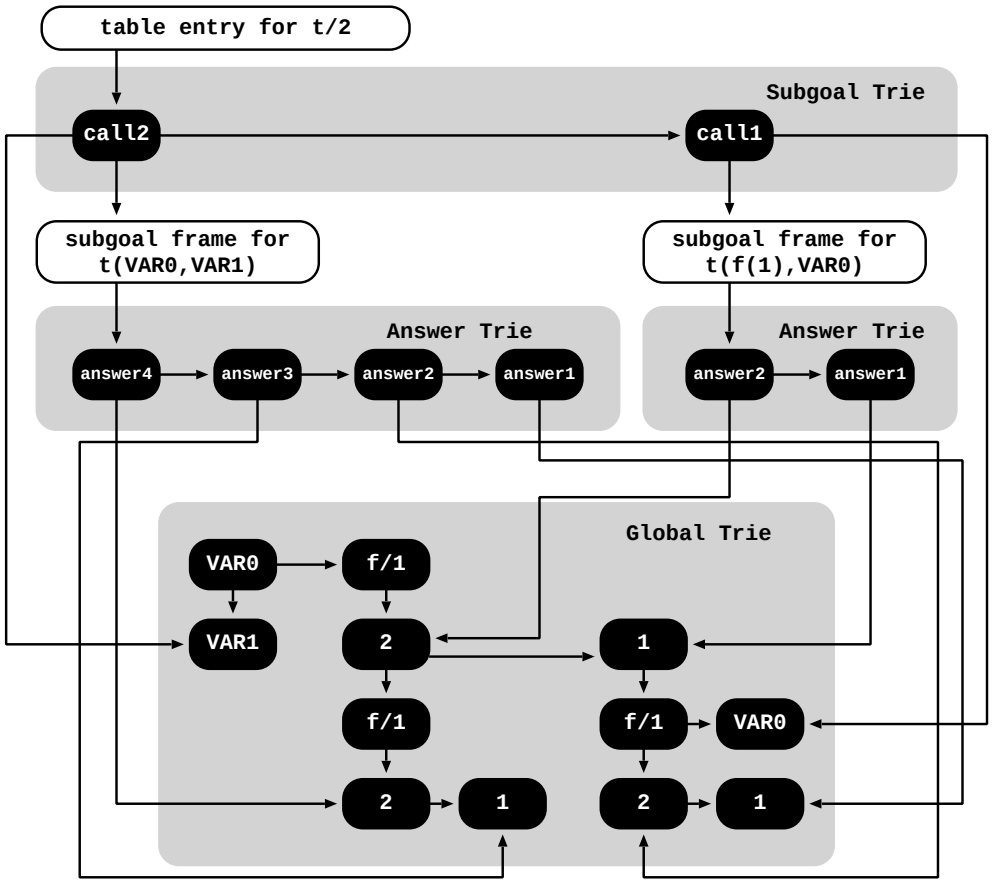



Figure 4.2: YapTab's table organization using the GT-CA design.

`call1` in Fig. 4.2). The `call1` node serves two purposes: its token's field is used to store the reference to the leaf node of the GT-CA's inserted path and its child field is used to store the reference to the corresponding subgoal frame. Afterwards, for the second subgoal call `t(X,Y)`, we start by inserting the call in the GT-CA and for that we represent the free variables `X` and `Y` by the nodes `VAR0` and `VAR1`, respectively. Next, we store a node in the subgoal trie (node `call2`) to represent the path inserted in the GT-CA.

For each answer, its term representation is inserted first in the GT-CA and then we stored a node in the corresponding answer trie, to represent the path inserted in the GT-CA (nodes labeled `answer1`, `answer2`, `answer3`, `answer4` in Fig. 4.2). Notice that in some situations, only part (or possibly none) of the term construction in the GT-CA is required, if part or the complete term representation already exists, thus emphasizing the contributions of a GT to store all term representations.

With this example, we can also see that with the GT-CA we cannot share the representation of common terms appearing at different arguments or substitution positions. An example is

the representation of the terms `f(1)`, `f(2)` and `VAR0`, which appear more than once in the GT. In fact, a subgoal call is represented by a sequence of argument terms while an answer is represented by a sequence of substitution terms. Moreover, when the number of argument or substitution terms is greater than one, the representation of a subgoal call or answer can end at internal nodes of other subgoal calls and/or answers, and not necessarily at a leaf node. This specific situation raises difficulties when supporting table abolish operations, since individual nodes can be part of different subgoal calls and/or answers representation. In this case the removal process of a individual node can not be done while it belongs to other different term representations. This problem can be solved by introducing an extra field in each trie node to count the number of paths it belongs to and only allowing deletion when it reaches zero, but this solution is contradictory with the GT goal of saving memory usage.

Another drawback of the GT-CA design occurs when a subgoal is completed. As mentioned previously, a strategy exists to avoid answer recovery using bottom-up unification and performing instead what is called a completed table optimization [8]. This optimization implements answer recovery by top-down traversing the completed answer trie and by executing specific WAM-like code from the answer trie nodes. However, when traversing the GT-CA with a top-down approach, traversed nodes can belong to several different subgoal and/or answer tries. So, with the GT-CA approach this optimization is no longer possible.

## 4.2 Global Trie for Terms

The *Global Trie for Terms* (GT-T) design can be seen as an extension of the previous approach [33]. The GT-T was designed to optimize the GT structure organization by maximizing the sharing of tabled data which is structurally equal. In the GT-T design, all argument and substitution terms appearing in tabled subgoal calls and/or answers are represented only once in the common GT, this allows to prevent situations where argument and substitution terms are represented more than once as in the example of Fig. 4.2.

As an extension of the previous GT-CA design, the GT-T data structure is still a tree structure. However, in this organization, each different path through the trie nodes represents a unique argument and/or substitution term, in contrast to the previous strategy where a path could represent more than an argument or substitution term. Therefore, the representation of terms always end at leaf trie nodes. In this table organization, the subgoal and answer tries data structure are no longer represented as a unique level of trie nodes. In both tries, each path is now composed of a fixed number of trie nodes, representing in the subgoal trie the arguments for the tabled subgoal call, or representing the substitution terms in the answer trie. More specifically, for the subgoal tries, each node now represents an argument term

in which the node's token is used to store the reference to the unique path in the GT-T where the actual argument term is represented. Similarly for the answer tries, each node now represents a substitution term, where the node's token stores the reference to the path's leaf node in the GT-T. The features used in tries to maintain the list of sibling nodes and to enable answer recovery in insertion order, introduced by YapTab's original subgoal and answer tries representation, remains unaltered.

Figure 4.3 illustrates how the GT-T design works, by stressing its most important features, and for that we use again the example from Fig. 4.1. Initially, the subgoal trie and the GT-T are empty. Then, the first subgoal `t(f(1),X)` is called and the two argument terms, `f(1)` (represented by the tokens `f/1` and `1`) and `X` (token `VAR0`), are first inserted in the GT-T. Afterwards, the argument terms are represented in the subgoal trie by two nodes (nodes `arg1` and `arg2`), and each node's token stores the reference to the leaf node of the corresponding term representation inserted in the GT-T. For the second subgoal call `t(X,Y)`, the argument terms `VAR0` and `VAR1`, representing respectively `X` and `Y`, are also first inserted in the GT-T, followed by the insertion of two nodes in the subgoal trie to represent them. In each token's node we store the reference to the corresponding representation in the GT-T.

When processing answers, the procedure is similar to the one executed for subgoal calls. For each substitution term, we also insert first its representation in the GT-T and then we insert a node in the corresponding answer trie, in order to store the reference to its path in the GT-T (nodes labeled `subs1` and `subs2` in Fig. 4.3). As shown in Fig. 4.3, the substitution terms for the complete set of answers for the two subgoal calls only include the terms `f(1)` and `f(2)`. Moreover, as `f(1)` was inserted in the global trie at the time of the first subgoal call, we only need to insert `f(2)` (represented by the nodes `f/1` and `2`), meaning that in fact we only need to insert the token `2`, in order to represent the full set of answers. So, we are maximizing the sharing of common terms appearing at different arguments or substitution positions. For this particular example, the result is a very compact representation of the GT, as most subgoal calls and/or answers share the same term representations.

Regarding space reclamation, as each different path in the GT-T always ends at a leaf node, we can use the child field (that is always NULL in a leaf node) to count the number of references to the path it represents. This feature is of uttermost importance for the deletion process of a path, which can only be performed when there is no reference to it, this is true when the leaf node's child field reaches zero. With this feature, the previous GT-CA's problem of supporting table abolish operations without introducing extra memory overheads, is solved.

Another GT-CA's problem was related with compiled tries, i.e., the technique used on completion of a subgoal. With GT-T such problem no longer exists and in order to enable the necessary topdown traversing, we keep the GT only with the term representations and store

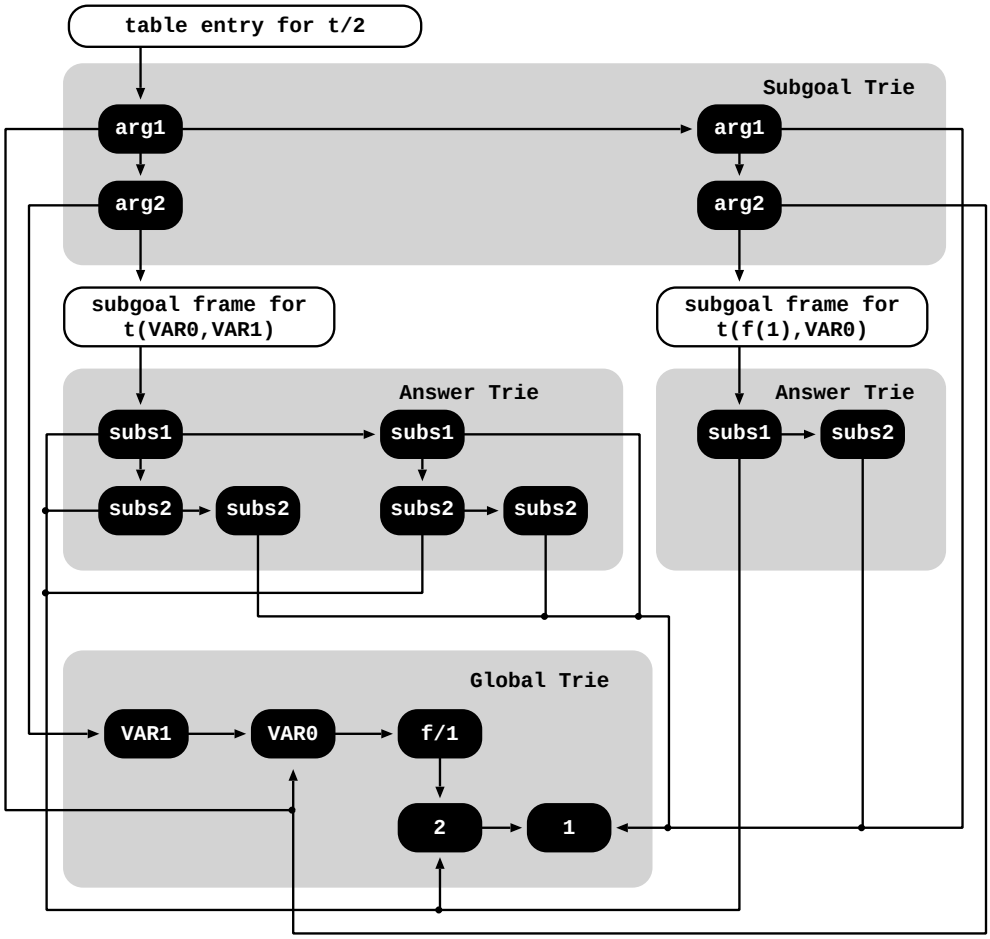



Figure 4.3: YapTab's table organization using the GT-T design.

the WAM-like instructions in the answer tries, as in the original design [8]. The difference caused by the existence of the GT is a new set of high-level WAM-like instructions, i.e., instead of working at the level of atoms/terms/functors/lists as in [8], each instruction works at the level of the substitution terms. For example, considering again the loading of four answers for the call `t(X,Y)`, one has two choices for the variable `X` and, to each variable `X`, we have two choices for variable `Y` (combination between two variables). In the GT-T design, the answer trie nodes representing the choices for `X` and for `Y` (nodes `subs1` and `subs2` respectively) are compiled with a WAM-like sequence of trie instructions, such as `try_subs_term` (for first choices) and `trust_subs_term` (for second/last choices). GT-T's compiled tries also include a `retry_subs_term` instruction (for intermediate choices) and a `do_subs_term` instruction (for single choices).

## 4.3 Global Trie for Subterms

In this design, we optimize the GT-T in order to obtain higher efficiency at the memory level. The *Global Trie for Subterms* (GT-ST) maintains most of the GT-T features, such as the sharing of the tabled data that is structurally equal. Yet, in this last design, we take into account the use of tabling mechanisms in real world problems, which require extensive search and where redundant data commonly occur. Therefore, we maximize the representation of the structural equal data at a *second level*, by avoiding the representation of equal subterms, and thus preventing situations where the representation of those subterms occurs more than once.

Although GT-ST uses the same tree structure for data structures, every different path can now represent a complete term or a subterm of another term, but still being an unique term. This particularity is evidenced in GT's compound term construction, such as lists or functors, that also have compound terms as arguments. In what follows, we will refer to compound terms arguments which are compound terms too, as subterms. In this case, we change the structure of the term in the GT by creating singular structures for each subterm, i.e., when inserting a term such as `f(p(1))`, after the construction of the functor `f/1` the insertion is stopped, and the construction of the subterm `p(1)` is inserted as a individual term in the GT. After the complete insertion of subterm `p(1)`, the construction of the main term is resumed by inserting a node pointing to the respective subterm representation previously made.

Although the structural differences in the GT-ST table space design, GT-T's structure for subgoal and answer tries, where each path is composed by a fixed number of nodes representing, respectively, the arguments for table subgoal calls or the substitution terms, is used without changes. Thus the subgoal trie and answer trie nodes store the pointers to the respective representation in the GT. Features regarding the subgoal frame structure, such as to maintain the chronological order of answer's insertion and correct recovery, also remain unaltered.

One last optimization is provided in GT-ST design, which can be also applied to the previous GT-CA and GT-T designs. The goal is to prevent the single node term representation in the GT, such as when representing atoms, integers and variables, by inserting them in the respective subgoal or answer trie, thus preventing unnecessary memory usage. The procedure consists in inserting directly the subgoal call arguments or substitution terms, which have a single node representation, in the respective subgoal or answer trie, thus avoiding its representation in the global trie. This optimization is straightforward. Since, by default, we are inserting a node in the subgoal trie or answer trie to point to the respective representation in the GT, for atomic terms we now avoid this and use the node to store the respective term. Figure 4.4 shows an example of how the previous GT-T design stores subterms by illustrating the resulting table data structures for the program described in the top of the figure.

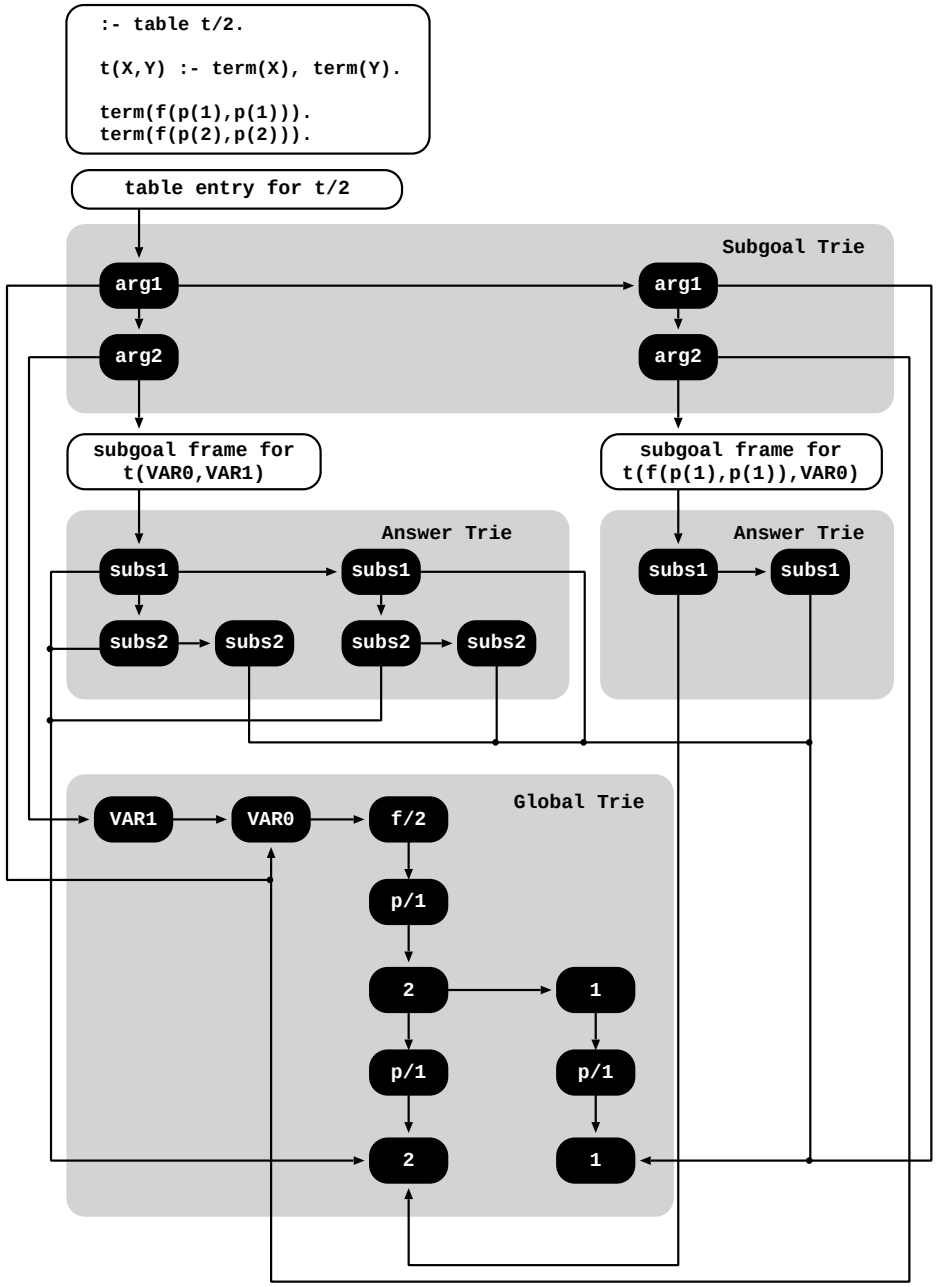



Figure 4.4: YapTab's table organization for compound terms using the GT-T design.

Figure 4.5 illustrates GT-ST design behavior, using the same example from Fig. 4.4. Initially, the subgoal trie and GT-ST are empty. Next the first subgoal `t(f(p(1),p(1)),X)` is called and the two argument terms are inserted in the global trie. Regarding the insertion of the first argument, `f(p(1),p(1))`, we emphasize the differences between this and the previous GT-T design.

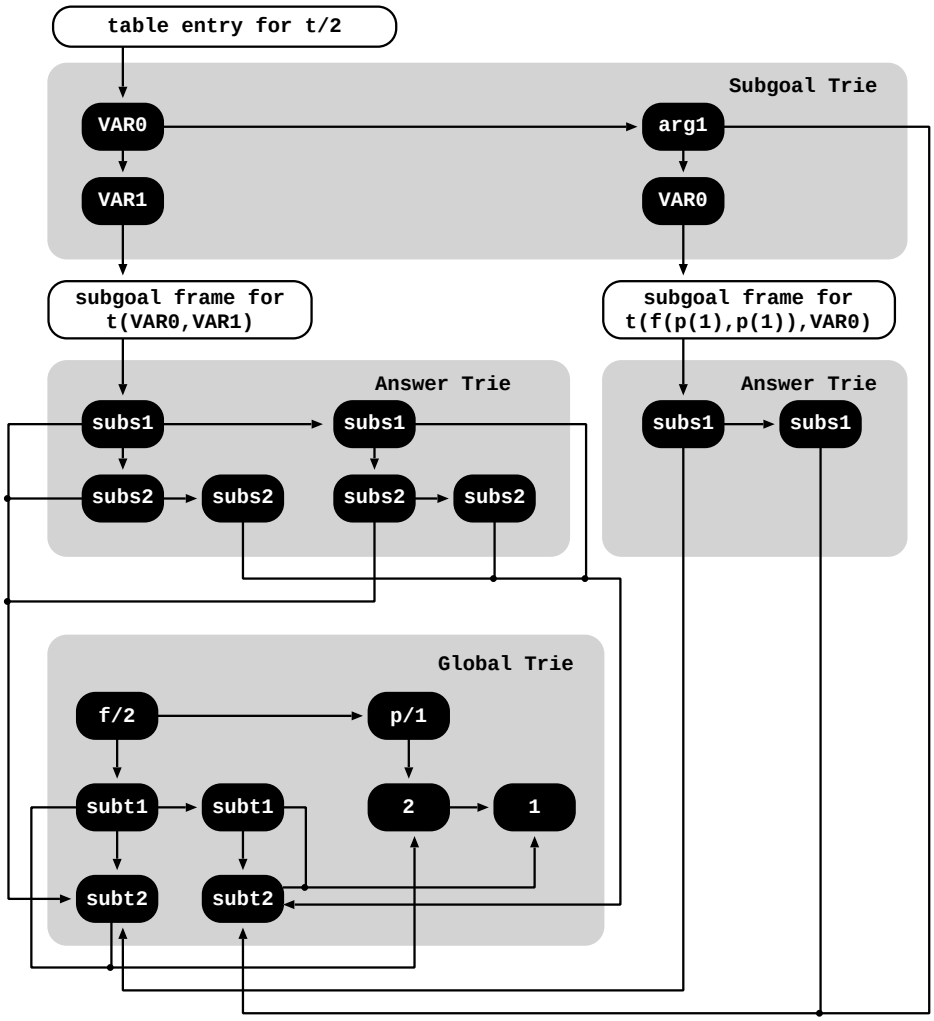



Figure 4.5: YapTab's table organization for compound terms using GT-ST.

Primarily, the node to represent the functor `f/2` is inserted, but then the insertion of functor `p/1` is stopped and the term `p(1)` is inserted as a distinct term in the GT-ST, i.e., as a sibling of the already stored node `f/2`. The nodes for `p/1` and `1` are then inserted in the GT-ST. Next, a node is inserted, in the place where we have previously stopped the term construction (as a child node of node `f/2`), to store the reference to the leaf node of the subterm `p(1)` representation. The construction of the main term then continues, applying an analogous procedure to the second argument of `f/2`. However the subterm `p(1)` is already stored in the GT, therefore it is only required the insertion of a node to store the reference to `p(1)` representation's leaf node. Afterwards, the respective argument node (node `arg1` in Fig. 4.5) is inserted in the subgoal trie storing the GT-ST reference representing `f(p(1),p(1))`. For the second subgoal call, `t(X,Y)`, we do not interact with the GT-ST. Therefore for each argument term, `X` and `Y`, we simply store in the subgoal trie the respective nodes with `VAR0`

and `VAR1` labels, as shown in the Fig. 4.5.

The procedure used when processing answers is equivalent to the one used when storing the subgoal call arguments. For each substitution term (if not an atomic term), we first insert the term in the GT-ST and then we store a node in the corresponding answer trie to store the reference to its path in the GT-ST (nodes labeled `subs1` and `subs2` in Fig. 4.5). In Fig. 4.5 the substitution terms for the complete set of answers for all subgoal calls are `f(p(1),p(1))` and `f(p(2),p(2))`. Thus, as `f(p(1),p(1))` is already stored in the global trie (inserted when storing the first subgoal call), we only need to store the second term in order to represent the whole set of answer. With this approach we increase the sharing of common subterms between terms and reduce the complexity when storing atomic terms.

Regarding space retrieval, the GT-ST design has the same features of the GT-T, where every path representing a singular term always ends at a leaf node. We also use the child field (that is always NULL in a leaf node) to count the number of references to it. This procedure works in any situation, even in what concerns to subterm's referencing. As mentioned in the previous section, regarding the support of table abolish operations, this feature is of uttermost importance in the deletion of a path, which occurs when the child's node field is zero. As provided in the GT-T design, the GT-ST also supports the techniques used on completion of a subgoal, keeping the global trie only with the term representation and storing the WAM-like instructions in the answer tries. Although, in this design we use an hybrid set of WAM-like instructions, ones that work at the level of the substitution terms and other that work at level of the atomic terms. Therefore, taking into consideration the position of the node in the answer trie and if it is a compound term or an atomic term. Hence, answer trie nodes are compiled with the instructions: `try_subs_term/atom` for first choices, `retry_subs_term/atom` for intermediate choices, `trust_subs_term/atom` for last choices and `do_subs_term/atom` for single choices.

# Chapter 5

# Implementation

In this chapter, we focus on the implementation details for YapTab's alternative table designs and we describe the GT data structures and algorithms in more detail. Throughout, we also describe how tries are structured, specifying the main features of trie nodes, and present the main procedures which interact with tries, preforming comparisons with YapTab's original table design. In what follows, we describe the three previously presented alternatives, detailing them separately.

## 5.1 Global Trie for Calls and Answers

We next describe the first presented alternative to YapTab's table design. We start with Fig. 5.1 describing in more detail the table organization previously presented in Fig. 4.2 for the subgoal call `t(f(1),X)`. Internally, all tries are represented by a top root node, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the *root node* is stored in the corresponding table entry's `subgoal_trie_root_node` data field. For the answer tries, the root node is stored in the corresponding subgoal frame's `answer_trie_root_node` data field. For the global trie, the root node is stored in the `GT_ROOT_NODE` global variable. Regarding the trie nodes, remember that they are internally implemented as 4-field data structures. The first field (entry) stores the token for the node and the second (child), third (parent) and fourth (sibling) fields store pointers, respectively, to the first child node, to the parent node, and to the sibling node. Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. Given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`. Figure 5.2 shows the pseudo-code for this procedure.

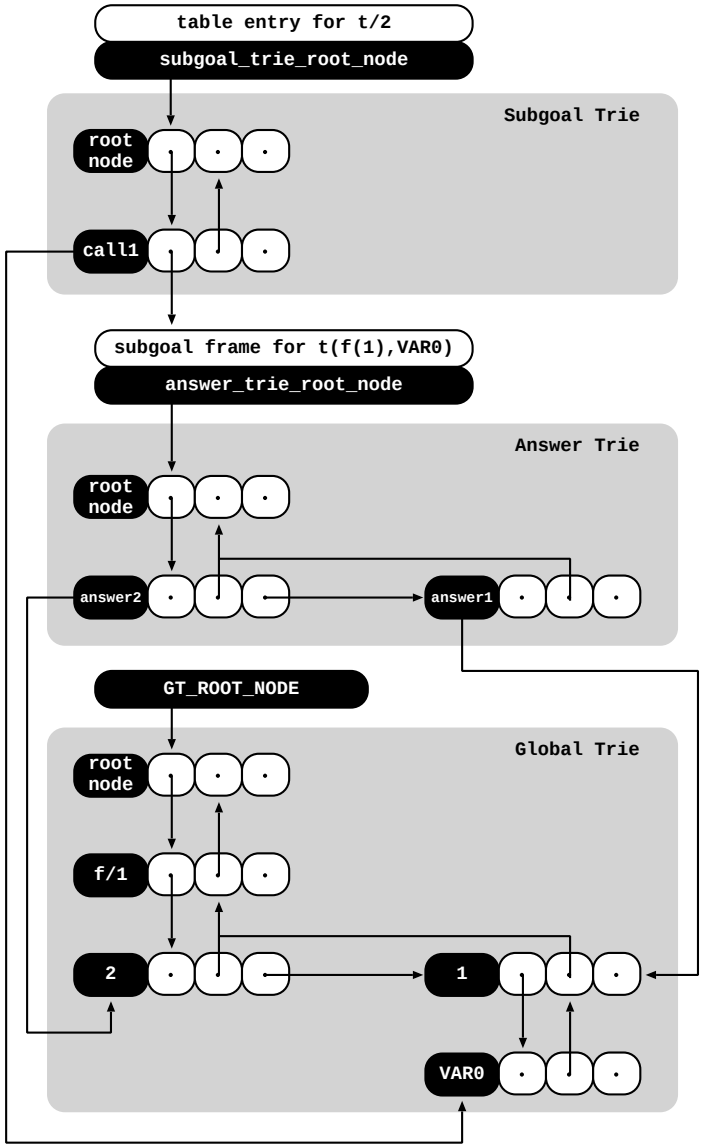



Figure 5.1: Implementation details for the GT-CA design.

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given parent node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the token, child, parent and sibling fields of the new trie node.

Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. Searching through a list of sibling nodes is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (`MAX_SIBLING_NODES_PER_LEVEL`) controls whether to dynamically index the nodes

```
trie_node_check_insert(TRIE_NODE parent, TOKEN t) {
  child = parent->child
  if (child == NULL) {              // the list of sibling nodes is empty
    child = new_trie_node(t, NULL, parent, NULL)
    parent->child = child
  } if (is_not_a_hash_table(child)) {    // sibling nodes without hashing
    sibling_nodes = 0             // to count the number of sibling nodes
    do {           // check if token t is already in the list of siblings
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    } while (child)
    child = new_trie_node(t, NULL, parent, parent->child)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_LEVEL) {  // alloc new hash
      hash = new_hash_table(child)
      parent->child = hash
    } else
      parent->child = child
  } else {                                  // sibling nodes with hashing
    hash = child
    bucket = hash_function(hash, t)    // get the hash bucket for token t
    child = bucket -> child
    sibling_nodes = 0
    while (child) {     // check if token t is already in the hash bucket
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    }
    child = new_trie_node(t, NULL, parent, bucket)
    bucket -> child = child
    if (sibling_nodes > MAX_SIBLING_NODES_PER_BUCKET)      // expand hash
      expand_hash_table(hash)
  }
  return child
}
```

Figure 5.2: Pseudo-code for the `trie_node_check_insert()` procedure.

through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables when a second threshold value (`MAX_SIBLING_NODES_PER_BUCKET`) is reached for a particular hash bucket. If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given parent node. If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the

list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we use two interface procedures. For traversing a trie to check/insert for new calls or for new answers we use the

```
trie_check_insert(TRIE_NODE root, TERM t)
```

procedure, where `root` is the root node of the trie to be used and `t` is the call/answer term to be inserted. The `trie_check_insert()` procedure invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given term and returns the reference to the leaf node representing its path. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

To load a term from a trie back to the Prolog engine we use the

```
trie_load(TRIE_NODE leaf)
```

procedure, where `leaf` is the reference to the leaf node of the term to be returned. When loading a term, the trie nodes are traversed in bottom-up order. When inserting terms in the table space we need to distinguish two situations: **(i)** inserting tabled calls in a subgoal trie structure; and **(ii)** inserting answers in a particular answer trie structure. The former situation is handled by the `subgoal_check_insert()` procedure as shown in Fig. 5.3 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Fig. 5.4.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call) {
  st_root_node = te->subgoal_trie_root_node
  if (GT_ROOT_NODE) {                              // GT-CA table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, call)
    leaf_st_node = trie_node_check_insert(st_root_node, leaf_gt_node)
  } else {                                        // original table design
    leaf_st_node = trie_check_insert(st_root_node, call)
  }
  return leaf_st_node
}
```

Figure 5.3: Pseudo-code for the GT-CA's `subgoal_check_insert()` procedure.

In the original table design, the `subgoal_check_insert()` procedure simply uses the

`trie_check_insert()` procedure to check/insert the given call in the subgoal trie corresponding to the given table entry `te`. In the new design based on the GT-CA, the `subgoal_check_insert()` procedure now first checks/inserts the given call in the GT. Then, it uses the reference to the GT's leaf node representing call (`leaf_gt_node` in Fig. 5.3) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is still initialized as described before.

The `answer_check_insert()` procedure works similarly. In the original table design, it checks/inserts the given answer in the answer trie corresponding to the given subgoal frame `sf`. In the new design based on the GT-CA, it first checks/inserts the given answer in the GT and, then, it uses the reference to the GT's leaf node representing answer (`leaf_at_node` in Fig. 5.4) as the token to be checked/inserted in the answer trie corresponding to the given subgoal frame `sf`. Again, if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer) {
  at_root_node = sf->answer_trie_root_node
  if (GT_ROOT_NODE) {                                  // GT-CA table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, answer)
    leaf_at_node = trie_node_check_insert(at_root_node, leaf_gt_node)
  } else {                                            // original table design
    leaf_at_node = trie_check_insert(at_root_node, answer)
  }
  return leaf_at_node
}
```

Figure 5.4: Pseudo-code for the GT-CA's `answer_check_insert()` procedure.

Finally, the `answer_load()` procedure is used to consume answers. Figure 5.4 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie the answer given by the trie node `leaf_at_node`. In the new design based on the GT-CA, the `answer_load()` procedure first accesses the GT's leaf node (`leaf_gt_node` in Fig 5.5) represented in the token field of the given trie node (`leaf_at_node` in 5.5). Then, it uses the `trie_load()` procedure to load from the GT back to the Prolog engine the answer represented by the obtained GT's leaf node.

## 5.2 Global Trie for Terms

We now describe in more detail the GT-T data structures and algorithms. We start with Fig. 5.6 showing in more detail the table organization previously presented in Fig. 4.3

```
answer_load(ANSWER_TRIE_NODE leaf_at_node) {
  if (GT_ROOT_NODE) {                                  // GT-CA table design
    leaf_gt_node = leaf_at_node->token
    answer = trie_load(leaf_gt_node)
  } else                                             // original table design
    answer = trie_load(leaf_at_node)
  return answer
}
```

Figure 5.5: Pseudo-code for the GT-CA's `answer_load()` procedure.

for the subgoal call `t(X,Y)`. As mentioned previously, tries are represented by a top root node, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the root node is stored in the corresponding table entry's `subgoal_trie_root_node` data field. For the answer tries, the root node is stored in the corresponding subgoal frame's `answer_trie_root_node` data field. For the global trie, the root node is stored in the `GT_ROOT_NODE` global variable.

In this table organization, the trie nodes have the same structure as in the previous design, being internally implemented as 4-field data structures. The first field (token) stores the token for the node and the second (child), third (parent) and fourth (sibling) fields store pointers, respectively, to the first child node, to the parent node, and to the next sibling node. Remember that for the global trie, the leaf node's child field is used to count the number of references to the path it represents. For the answer tries, an additional field (code) is used to support compiled tries. As mentioned before, traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. Given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`.

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given parent node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the token, child, parent and sibling fields of the new trie node. For answer trie nodes, the code field is computed later when completion takes place. Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table.

As in the previous design, two threshold values, `MAX_SIBLING_NODES_PER_LEVEL` and `MAX_SIBLING_NODES_PER_BUCKET`, control whether to dynamically index/expand the nodes through a hash table. If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node

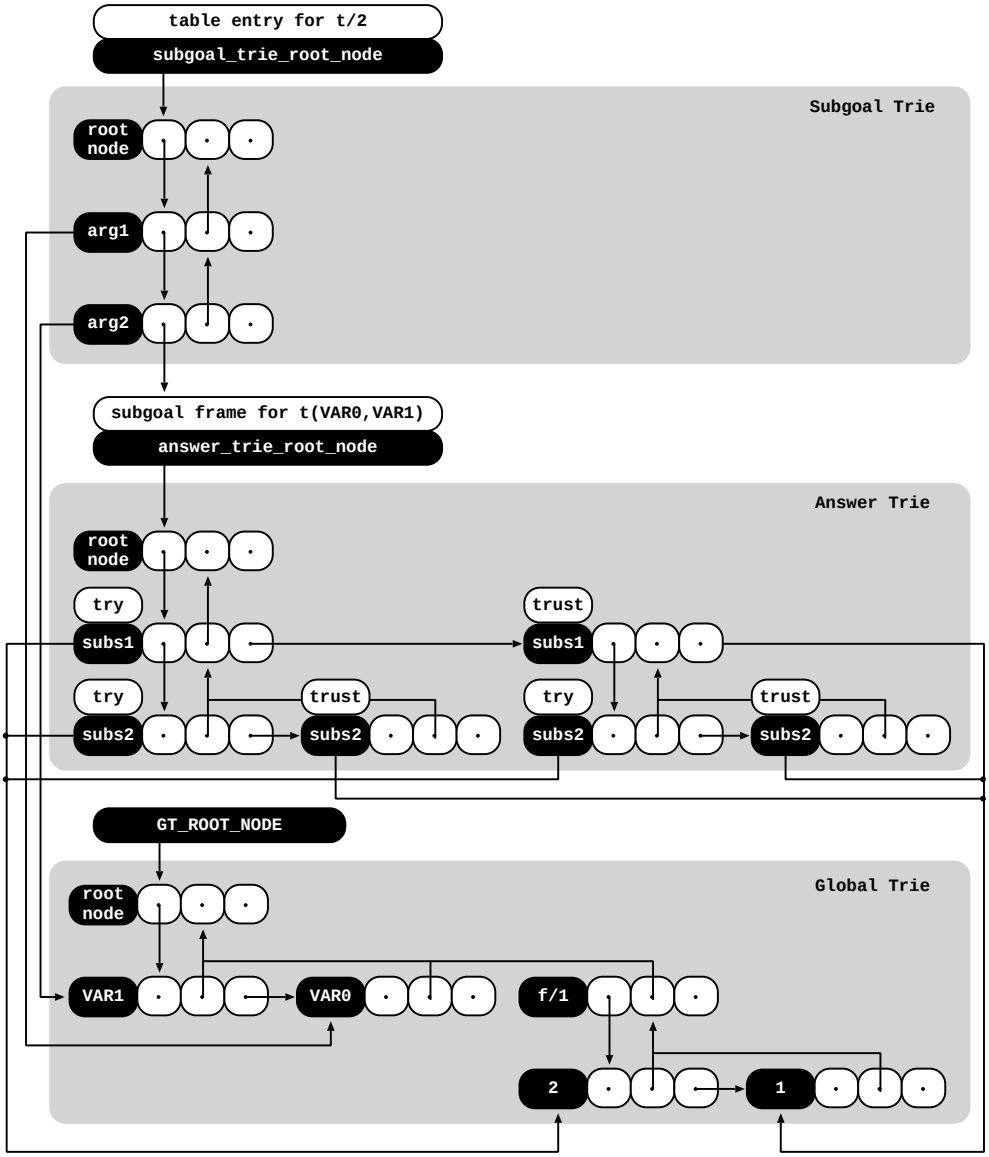



Figure 5.6: Implementation details for GT-T design.

is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given parent node. If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we still use two interface procedures:

```
trie_check_insert(TRIE_NODE root, TERM t)
trie_load(TRIE_NODE leaf)
```

Once more, the `trie_load()` is used to load a term from a trie back to the Prolog engine, where `leaf` is the reference to the leaf node of the term to be loaded. The `trie_check_insert()` is used for traversing a trie to check/insert for new terms, where `root` is the root node of the trie to be used and `t` is the term to be inserted. It invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given term `t` and returns the reference to the leaf node representing its path. Inserting tabled calls in a subgoal trie structure is now handled by the `subgoal_check_insert()` procedure as shown in Fig. 5.7 and inserting answers in a particular answer trie structure is now handled by the `answer_check_insert()` procedure as shown in Fig. 5.8.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call, SUBGOAL_ARITY a) {
  if (GT_ROOT_NODE) {                                  // GT-T table design
    st_node = te->subgoal_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_argument_term(call, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++       // increase number of paths it represents
      st_node = trie_node_check_insert(st_node, leaf_gt_node)
    }
    leaf_st_node = st_node
  } else                                             // original table design
    leaf_st_node = trie_check_insert(te->subgoal_trie_root_node, call)
  return leaf_st_node
}
```

Figure 5.7: Pseudo-code for the GT-T's `subgoal_check_insert()` procedure.

In the GT-T design, for each argument term `t`, the `subgoal_check_insert()` first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 5.7) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

The `answer_check_insert()` procedure works similarly. In the GT-T design, for each substitution term `t`, it first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 5.8) as the token to be checked/inserted in the answer trie corresponding to the given subgoal frame `sf`. Again,

if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer, SUBSTITUTION_ARITY a) {
  if (GT_ROOT_NODE) {                                    // GT-T table design
    at_node = sf->answer_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_substitution_term(answer, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++          // increase number of paths it represents
      at_node = trie_node_check_insert(at_node, leaf_gt_node)
    }
    leaf_at_node = at_node
  } else                                                 // original table design
      leaf_at_node = trie_check_insert(sf->answer_trie_root_node, answer)
  return leaf_at_node
}
```

Figure 5.8: Pseudo-code for the GT-T's `answer_check_insert()` procedure.

Finally, Fig. 5.9 shows the pseudo-code for the new `answer_load()` procedure. In the new GT-T design, for each answer trie node `at_node`, now the `answer_load()` procedure uses the `trie_load()` procedure to load from the GT-T back to the Prolog engine the substitution term given by the reference (`leaf_gt_node` in Fig. 5.9) stored in the corresponding token field.

```
answer_load(ANSWER_TRIE_NODE leaf_at_node, SUBSTITUTION_ARITY a) {
  if (GT_ROOT_NODE) {                                    // GT-T table design
    at_node = leaf_at_node
    for (i = a; i >= 1; i--) {
      leaf_gt_node = at_node->token
      t = trie_load(leaf_gt_node)
      put_substitution_term(t, answer)
      at_node = at_node->parent
    }
  } else                                                 // original table design
    answer = trie_load(leaf_at_node)
  return answer
}
```

Figure 5.9: Pseudo-code for the GT-T's `answer_load()` procedure.

## 5.3 Global Trie for Subterms

Finally, we then describe the data structures and algorithms for the GT-ST table design. Figure 5.10 shows in more detail the table organization previously presented in Fig. 4.5 for

the subgoal call `t(X,Y)`. As mentioned in the previous sections, tries are represented by a top root node, acting as the entry point for the corresponding trie data structure, and trie nodes are internally implemented as 4-field data structures. The first field (entry) stores the token for the node and the second (child), third (parent) and fourth (sibling) fields store pointers, respectively, to the first child node, to the parent node, and to the sibling node. Traversing a trie to check/insert for new calls or new answers is also implemented by repeatedly invoking a `trie_node_check_insert()` procedure for each token that represents the call/answer being checked. The same algorithm is applied on this design, i.e., given a trie node `parent` and a token `t`, the `trie_node_check_insert()` procedure returns the child node of `parent` that represents the given token `t`.

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given parent node.Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. The usage of the threshold values `MAX_SIBLING_NODES_PER_LEVEL` and `MAX_SIBLING_NODES_PER_BUCKET` remains unaltered. When using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of corresponding the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET` the current hash table is expanded. If not using hashing, the list of sibling nodes is sequentially traversed to be checked for one representation of the given token `t`. The procedure is stopped when such representation is found, returning the respective node. Otherwise, a new trie node inserted in the list of siblings. When the threshold value `MAX_SIBLING_NODES_PER_LEVEL` is reached, during a new node's insertion, a new hash table is initialized, inserting the new node in the new hash table.

As for the previous designs, in the GT-ST we also use two interface procedures to manipulate tries.

```
trie_check_insert(TRIE_NODE root, TERM t)
trie_load(TRIE_NODE leaf)
```

The `trie_load()` procedure is used to load a term from a trie back to the Prolog engine, where `leaf` is the reference to the leaf node of the term to be loaded. The `trie_check_insert()` is used for traversing a trie to check/insert for new terms, where `root` is the root node of the trie to be used and `t` is the term to be inserted. As described in the previous sections, the two distinct situations of inserting tabled calls in a subgoal trie structure and inserting answers in a particular answer trie structure are handled respectively by the `subgoal_check_insert()` and `answer_check_insert()` procedures. In the GT-ST

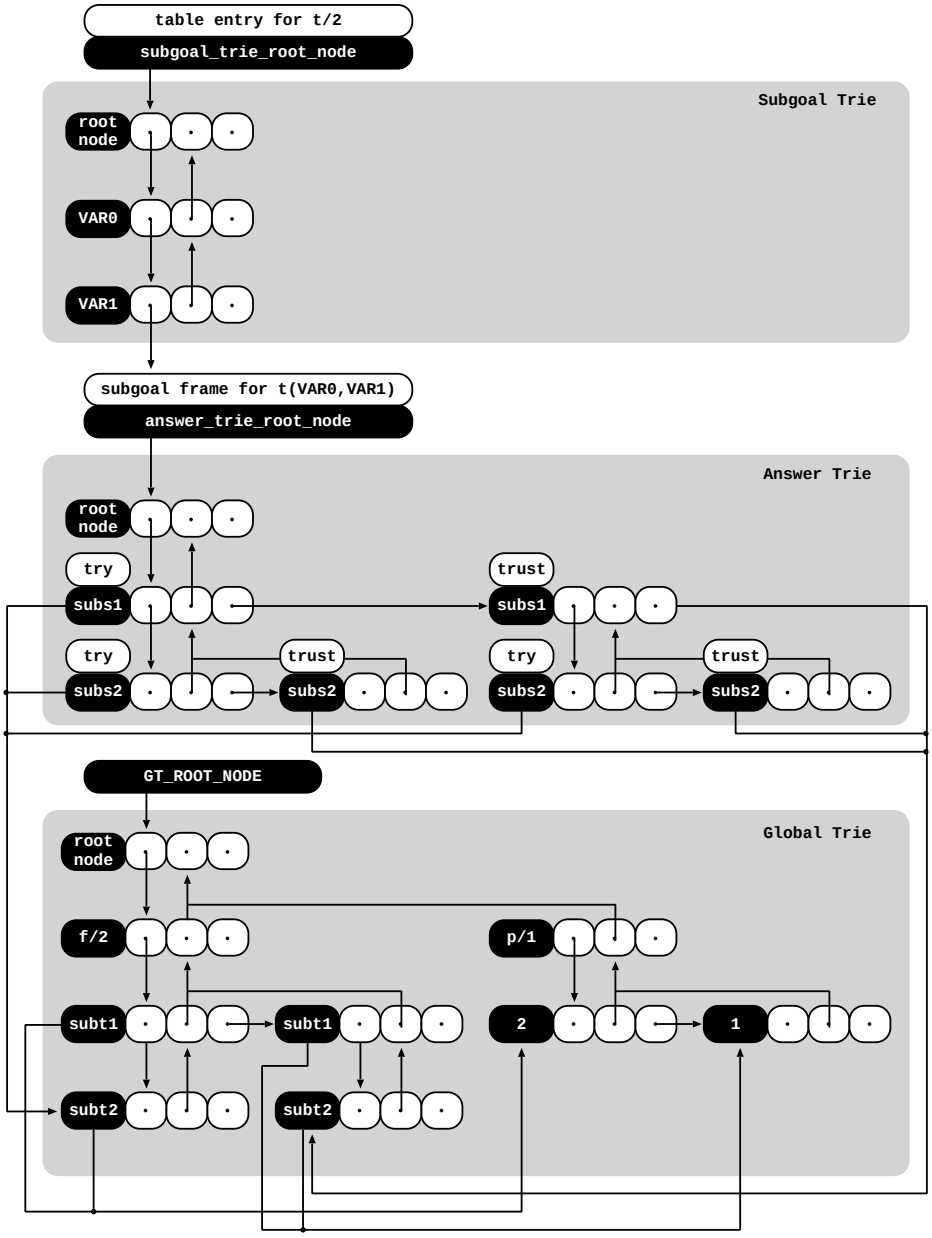



Figure 5.10: Implementation details for the GT-ST design.

design these procedures are analogues to the ones presented for the GT-T design, as shown respectively in Fig. 5.7 and Fig. 5.8. Both procedures start by first checking/inserting the term `t` in the GT, in order to use the reference to the leaf node representing `t` in the GT-T, as the token to be checked/inserted in the corresponding subgoal or answer trie. In the GT-ST, both procedures behave in same way in what regards to the subgoal and answer check/insert procedure.

The difference relies in the insertion of terms in the GT, and for that we have changed the

`trie_check_insert()` procedure in such a way that when a compound term has a compound term as an argument, the procedure calls itself. In what remains we will refer to a term argument as a subterm. Figure 5.11 shows the pseudo-code for the changes made to the `trie_check_insert()` procedure in order to support the new algorithm.

```
trie_check_insert(TRIE_NODE root, TERM t) {
  current_node = root
  if (is_atomic_term(t)) {
    current_node = trie_node_check_insert(current_node, t)
  } else if (is_compound_term(t)) {                    // GT-ST table design
    if (current_node == GT_ROOT_NODE) {
      st = compound_term_name(t)
      a = compound_term_arity(t)
      current_node = trie_node_check_insert(current_node, st)
      for (i = 0; i < a; i ++) {
        st = get_argument_term(t, i)
        current_node = trie_check_insert(current_node, st)
      }
    } else {                            // compound subterm of a compound term
      ref = trie_check_insert(GT_ROOT_NODE, t)
      current_node = trie_node_check_insert(current_node, ref)
    }
  }
  ...
  return current_node
}
```

Figure 5.11: Pseudo-code for the GT-ST's `trie_check_insert()` procedure for the GT-ST design.

Remember that, with the GT enabled, the `trie_check_insert()` procedure for a call or answer is called with the `GT_ROOT_NODE` as the `root` argument. For the given term `t`, we initially verify its type in order to preform the respective action of insertion in the trie. When `t` is a compound term, two situations can occur: **(i)** if the `current_node` is the `GT_ROOT_NODE` then the insertion proceeds by first inserting the term's name with the `trie_node_check_insert()` and then, for each element of `t` (subterm), by invoking the `trie_check_insert()` procedure; **(ii)** on the other hand, if the `current_node` is not the `GT_ROOT_NODE`, which means that `t` is an argument from a compound term, then we first call the `trie_check_insert()` procedure with the `GT_ROOT_NODE` and the term `t` as arguments. By doing that, `t` is inserted as a simple term in the GT and when the `trie_check_insert()` procedure returns, the reference `ref` to the leaf node of the subterm's path representation of `t` in the GT is inserted after the `current_node` by calling the `trie_node_check_insert()` procedure.

The `answer_load()` procedure in this design is used as in the GT-T design, i.e., it uses the `trie_load()` procedure to load from the GT back to the Prolog engine the substitution term given by the reference stored in the corresponding token field. In the case of subterm

references in the GT, the `trie_load()` procedure calls itself to first load the subterm reference from the GT.

Finally, in what regards the optimization mentioned in the previous chapter for the representation of atomic terms (integers, atoms and variables), Fig. 5.10 presents the changes made to the table space when using this optimization. Remember that in the previous GT design, when inserting an atomic term, being it part of an answer or subgoal, the algorithm first checks/inserts the term in the GT and only then inserts the reference to its representation in the respective subgoal or answer trie. To implement this optimization, the `subgoal/answer_check_insert()` procedures were slightly changed. Before we insert a term `t` in the GT, now we first verify if it is an atomic term, and if so, instead of inserting it in the GT we represent the term in its respective subgoal/answer trie. Figure 5.12 shows the pseudo-code for this optimization applied to the `subgoal_check_insert()` procedure. It is applied similarly to the `answer_check_insert()` procedure.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call, SUBGOAL_ARITY a) {
  if (GT_ROOT_NODE) {                                          // GT-ST table design
    st_node = te->subgoal_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_argument_term(call, i)
      if (is_atomic_term(t))                              // atomic term optimization
        st_node = trie_node_check_insert(st_node, t)
      else {
        leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
        leaf_gt_node->child++           // increase number of paths it represents
        st_node = trie_node_check_insert(st_node, leaf_gt_node)
      }
    }
    leaf_st_node = st_node
  } else                                                     // original table design
    leaf_st_node = trie_check_insert(te->subgoal_trie_root_node, call)
  return leaf_st_node
}
```

Figure 5.12: Pseudo-code for the GT'ST `subgoal_check_insert()` procedure optimized for atomic terms.

# Chapter 6

# Experimental Results

In this chapter we present the experimental results obtained for the optimizations previously described, and for that we compare the running times and system's memory spent by each. Initially, we present results for the new compact list terms representation, making comparison with the YapTab system. Afterwords, we discuss the results obtained by testing the GT table designs, once more comparing then with the YapTab system. The environment for our experiments was an Intel(R) Core(TM)2 Quad 2.66GHz with 3.2 GBytes of main memory and running the Linux kernel 2.6.24-28-generic with YapTab 6.2.0.

## 6.1 Compact List Terms

We next present some experimental results comparing YapTab with and without support for compact lists. To put the performance results in perspective, we have defined a top query goal that calls recursively a tabled predicate `list_terms/1` that simply stores in the table space list terms facts. We experimented the `list_terms/1` predicate using 50,000, 100,000 and 200,000 list terms of sizes 60, 80 and 100 for empty-ending and term-ending lists with the first and with the last element different. Tables 6.1 and 6.2 show the table memory usage (columns ***Memory***), in KBytes, and the running times, in milliseconds, to store (columns ***Store***) the tables (first execution) and to load from the tables (second execution) the complete set of answers without (columns ***Load***) and with (columns ***Comp***) compiled tries for YapTab using standard lists (column ***YapTab***) and using the final design for compact lists (column ***YapTab+CL/YapTab***). For compact lists, we only show the memory and running time ratios over YapTab using standard lists. The running times are the average of five runs.

The results in Tables 6.1 and 6.2 clearly confirm that the new trie design based on compact lists can decrease significantly memory usage when compared with standard lists.

| *Empty-Ending Lists* | *YapTab* | | | | *YapTab+CL/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* | *Memory* | *Store* | *Load* | *Comp* |
| *First element different* | | | | | | | | |
| **50,000 [$E_1$,...,$E_{60}$]** | 117,187 | 327 | 48 | 48 | **0.51** | **0.50** | **0.72** | **0.73** |
| **50,000 [$E_1$,...,$E_{80}$]** | 156,250 | 486 | 62 | 62 | **0.51** | **0.50** | **0.66** | **0.65** |
| **50,000 [$E_1$,...,$E_{100}$]** | 195,312 | 641 | 75 | 75 | **0.51** | **0.47** | **0.65** | **0.65** |
| **100,000 [$E_1$,...,$E_{60}$]** | 234,375 | 775 | 93 | 93 | **0.51** | **0.47** | **0.74** | **0.74** |
| **100,000 [$E_1$,...,$E_{80}$]** | 312,500 | 1,135 | 122 | 122 | **0.51** | **0.45** | **0.67** | **0.68** |
| **100,000 [$E_1$,...,$E_{100}$]** | 390,625 | 1,531 | 150 | 149 | **0.51** | **0.46** | **0.65** | **0.66** |
| **200,000 [$E_1$,...,$E_{60}$]** | 468,750 | 1,868 | 187 | 186 | **0.51** | **0.48** | **0.74** | **0.75** |
| **200,000 [$E_1$,...,$E_{80}$]** | 625,000 | 2,544 | 250 | 247 | **0.51** | **0.48** | **0.66** | **0.66** |
| **200,000 [$E_1$,...,$E_{100}$]** | 781,250 | 3,161 | 300 | 302 | **0.51** | **0.54** | **0.66** | **0.68** |
| *last element different* | | | | | | | | |
| **50,000 [$E_1$,...,$E_{60}$]** | 1,955 | 58 | 22 | 21 | **0.50** | **0.77** | **0.70** | **0.73** |
| **50,000 [$E_1$,...,$E_{80}$]** | 1,956 | 82 | 29 | 28 | **0.50** | **0.73** | **0.67** | **0.69** |
| **50,000 [$E_1$,...,$E_{100}$]** | 1,957 | 94 | 35 | 35 | **0.50** | **0.78** | **0.68** | **0.68** |
| **100,000 [$E_1$,...,$E_{60}$]** | 3,909 | 122 | 43 | 43 | **0.50** | **0.76** | **0.75** | **0.72** |
| **100,000 [$E_1$,...,$E_{80}$]** | 3,910 | 156 | 57 | 57 | **0.50** | **0.77** | **0.72** | **0.70** |
| **100,000 [$E_1$,...,$E_{100}$]** | 3,910 | 191 | 70 | 70 | **0.50** | **0.79** | **0.69** | **0.67** |
| **200,000 [$E_1$,...,$E_{60}$]** | 7,815 | 255 | 87 | 92 | **0.50** | **0.73** | **0.72** | **0.68** |
| **200,000 [$E_1$,...,$E_{80}$]** | 7,816 | 318 | 118 | 118 | **0.50** | **0.76** | **0.65** | **0.66** |
| **200,000 [$E_1$,...,$E_{100}$]** | 7,817 | 377 | 141 | 140 | **0.50** | **0.78** | **0.67** | **0.67** |

Table 6.1: Table memory usage (in KBytes) and store/load times (in milliseconds) for YapTab with and without support for compact lists for empty-ending lists with different first or last elements.

In particular, for empty-ending lists, with the first and with the last element different, and for term-ending lists with the first element different, the results show an average reduction of 50%. For term-ending lists with the last element different, memory usage is almost the same. This happens because the memory reduction obtained in the representation of the common list elements (respectively 59, 79 and 99 elements in these experiments) is residual when compared with the number of different last elements (50,000, 100,000 and 200,000 in these experiments).

Regarding running time, the results in Tables 6.1 and 6.2 indicate that compact lists can achieve impressive gains for storing and loading list terms. In these experiments, the storing time using compact lists is around 2 times faster for list terms with the first element different, and around 1.3 (0.79 ratio) to 1.4 (0.73 ratio) times faster for list terms with the last element different. Note that this is the case even for term-ending lists, where there is no significant memory reduction. This happens because the number of nodes to be traversed when navigating the trie data structures for compact lists is considerably smaller

| *Term-Ending Lists* | *YapTab* | | | | *YapTab+CL/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* | *Memory* | *Store* | *Load* | *Comp* |
| *1st element different* | | | | | | | | |
| **50,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 115,235 | 320 | 48 | 47 | **0.52** | **0.51** | **0.72** | **0.73** |
| **50,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 154,297 | 471 | 62 | 62 | **0.51** | **0.53** | **0.67** | **0.66** |
| **50,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 193,360 | 657 | 74 | 73.6 | **0.51** | **0.47** | **0.66** | **0.65** |
| **100,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 230,469 | 732 | 97 | 96 | **0.52** | **0.50** | **0.72** | **0.72** |
| **100,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 308,594 | 1149 | 124 | 122 | **0.51** | **0.46** | **0.66** | **0.67** |
| **100,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 386,719 | 1516 | 149 | 146 | **0.51** | **0.49** | **0.66** | **0.67** |
| **200,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 460,937 | 1853 | 187 | 190 | **0.52** | **0.52** | **0.77** | **0.74** |
| **200,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 617,188 | 2417 | 244 | 248 | **0.51** | **0.51** | **0.69** | **0.69** |
| **200,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 773,438 | 3152 | 296 | 299 | **0.51** | **0.53** | **0.67** | **0.66** |
| *last element different* | | | | | | | | |
| **50,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 979 | 57 | 22 | 22 | 1.00 | **0.82** | **0.70** | **0.74** |
| **50,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 980 | 74 | 28 | 28 | 1.00 | **0.89** | **0.69** | **0.69** |
| **50,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 981 | 94 | 43 | 39 | 1.00 | **0.79** | **0.54** | **0.59** |
| **100,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 1,956 | 113 | 42 | 42 | 1.00 | **0.84** | **0.74** | **0.74** |
| **100,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 1,956 | 146 | 56 | 60 | 1.00 | **0.81** | **0.64** | **0.69** |
| **100,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 1,957 | 190 | 74 | 70 | 1.00 | **0.77** | **0.62** | **0.68** |
| **200,000 [$E_1$,...,$E_{59}$\|$E_{60}$]** | 3,909 | 238 | 85 | 90.4 | 1.00 | **0.77** | **0.78** | **0.69** |
| **200,000 [$E_1$,...,$E_{79}$\|$E_{80}$]** | 3,910 | 294 | 113 | 113 | 1.00 | **0.85** | **0.73** | **0.67** |
| **200,000 [$E_1$,...,$E_{99}$\|$E_{100}$]** | 3,910 | 364 | 140 | 140 | 1.00 | **0.81** | **0.70** | **0.67** |

Table 6.2: Table memory usage (in KBytes) and store/load times (in milliseconds) for YapTab with and without support for compact lists for lists with different first or last elements.

than the number of nodes for standard lists. These results also indicate that compact lists can outperform standard lists for loading terms, both with and without compiled tries, and that the reduction on the running time seems to decrease as the size of the list terms being considered increases.

## 6.2 Global Trie

We next present some experimental results comparing YapTab with and without support for the common global trie data structure. To put the performance results in perspective and have a well-defined starting point comparing the GT-T and GT-ST approaches, first we have defined a tabled predicate `t/5` that simply stores in the table space terms defined by `term/1` facts, and then we used a top query goal `test/0` to recursively call `t/5` with all combinations of one and two free variables in the arguments. An example of such code for functor terms of arity 1 (1000 terms in total) is shown next.

```
:- table t/5.
t(A,B,C,D,E) :- term(A), term(B), term(C), term(D), term(E).

test :- t(A,f(1),f(1),f(1),f(1)), fail.
test :- t(f(1),f(1),f(1),f(1),A), fail.
test :- t(A,B,f(1),f(1),f(1)), fail.
...
test :- t(f(1),f(1),f(1),A,B), fail.
test.

term(f(1)).
term(f(2)).
term(f(3)).
...
term(f(998)).
term(f(999)).
term(f(1000)).
```

We experimented the `test/0` predicate with 10 different kinds of 1000 `term/1` facts: integers, atoms, functor (with arity 1, 2, 4 and 6) and list (with length 1, 2 and 4) terms. Table 6.3 shows the table memory usage (column ***Memory***), in MBytes, and the running times, in milliseconds, to store (column ***Store***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (column ***Load***) and with (column ***Comp***) compiled tries for YapTab's original table design.

Table 6.4 shows the same figures presented in Table 6.3, memory in MBytes and running times spent to store tables and to load answer from tables, with and without compiled tries, in milliseconds, but when using the GT-T (column ***GT-T/YapTab***) or the GT-ST (column ***GT-ST/YapTab***) designs. For this table, we only show the ratios over YapTab's original table design using the results presented in Table 6.3.

Notice, that the results obtained for the first GT design, the GT-CA, are not shown here, since this design is no longer supported and the changes made at YapTab's latest version did not include this design. Therefore, a fair comparison between all the GT designs is not possible. For reference, in Appendix A, we show the results obtained and published in [33] for a preliminary version of the GT-CA design.

The results in Table 6.4 suggest that both GT designs are a very good approach to reduce memory usage and that this reduction increases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for functor and list terms, the results show an increasing and very significant reduction on memory usage, for both GT-T and GT-ST approaches. The results for the special cases of integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the respective tries. Note that, although, integers and atoms terms are only represented in the respective tries, it is necessary to check for these type of term, in order to proceed with the respective store/load algorithm.

| *Terms* | *YapTab* | | | |
|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* |
| **1000 ints** | 191 | 1,270 | 345 | 344 |
| **1000 atoms** | 191 | 1,423 | 343 | 406 |
| **1000 f/1** | 191 | 1,680 | 542 | 361 |
| **1000 f/2** | 382 | 2,295 | 657 | 450 |
| **1000 f/4** | 764 | 3,843 | 973 | 631 |
| **1000 f/6** | 1,146 | 5,181 | 1,514 | 798 |
| **1000 [ ]/1** | 382 | 2,215 | 507 | 466 |
| **1000 [ ]/2** | 764 | 3,832 | 818 | 604 |
| **1000 [ ]/4** | 1,528 | 6,566 | 1,841 | 1,066 |

Table 6.3: Table memory usage (in MBytes) and store/load times (in milliseconds) for the `test/0` predicate using YapTab's original table design.

| *Terms* | *GT-T/YapTab* | | | | *GT-ST/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* | *Memory* | *Store* | *Load* | *Comp* |
| **1000 ints** | 1.00 | 1.05 | 1.00 | 1.00 | 1.00 | 1.09 | 1.11 | 1.07 |
| **1000 atoms** | 1.00 | 1.04 | 1.01 | 1.02 | 1.00 | 1.04 | 1.03 | 1.08 |
| **1000 f/1** | 1.00 | 1.32 | 1.16 | 2.10 | 1.00 | 1.34 | 1.17 | 2.13 |
| **1000 f/2** | **0.50** | 1.10 | 1.14 | 1.84 | **0.50** | 1.06 | 1.11 | 1.88 |
| **1000 f/4** | **0.25** | **0.81** | **0.98** | 1.44 | **0.25** | **0.78** | 1.04 | 1.53 |
| **1000 f/6** | **0.17** | **0.72** | **0.72** | 1.38 | **0.17** | **0.66** | **0.71** | 1.36 |
| **1000 [ ]/1** | **0.50** | 1.08 | 1.05 | 1.61 | **0.50** | 1.10 | 1.02 | 1.58 |
| **1000 [ ]/2** | **0.25** | **0.80** | **0.94** | 1.38 | **0.25** | 1.00 | 1.05 | 1.48 |
| **1000 [ ]/4** | **0.13** | **0.63** | **0.54** | **0.96** | **0.13** | **0.89** | **0.66** | 1.14 |
| ***Average*** | **0.53** | **0.95** | **0.95** | **1.42** | **0.53** | **0.99** | **0.99** | **1.47** |

Table 6.4: Table memory usage (in MBytes) and store/load times (in milliseconds) for the `test/0` predicate using YapTab with support for the common global trie data structure.

Regarding running time the results suggest that, in general, GT-ST, spends more time in the store and load term procedures. Such behaviour can be easily explained by the fact that, the GT-ST's storing and loading algorithms have more sub-cases to process in order to support subterms. These results also seem to indicate that memory reduction for small sized terms, generally comes at a price in storing time (between 4% and 32% more for GT-T and between 4% and 34% more for GT-ST in these experiments). The opposite occurs in the tests where term's length are higher (between 19% and 37% less for GT-T and 11% and 34% less for GT-ST). Note that with GT-T and GT-ST support, we pay the cost of navigating in two tries when checking/storing/loading a term. Moreover, in some situations, the cost of storing a new term in an empty/small trie can be less than the cost of navigating in the global trie, even when the term is already stored in the global trie. However, our results seem to suggest

that this cost decreases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for functor and list terms and functor terms, GT-T and GT-ST support showed to outperform the original YapTab design and, in particular, the reduction seems to decrease also proportionally to the length of the terms stored in the global trie.

The results obtained for loading terms also show some gains without compiled tries (around 5% for GT-T and 1% for GT-ST on average) but, when using compiled tries the results show some significant costs on running time (around 42% for GT-T and 47% for GT-ST on average). We believe that this cost is smaller for GT-T as a result of having less subcases in the storing/loading algorithms. On the other hand, we also believe that some cache behaviour effects, reduce the costs on running times, for both GT designs. As we need to navigate in the global trie for each substitution term, we kept accessing the same global trie nodes, thus reducing eventual cache misses. This seems to be the reason why for list terms of length 4, GT-T clearly outperforms the original YapTab design, both without and with compiled tries. Note that, for this particular case, the GT-T support only consumes 13% of the memory used in the original YapTab.

Next, we tested our approach with two well-known Inductive Logic Programming (ILP) [34] benchmarks: the *carcinogenesis* (***Carc***) and the *mutagenesis* (***Muta***) data sets. We used these data sets in a Prolog program that simulates the test phase of an ILP system. For that, first we ran the April ILP system [35] for the two data sets, each with two different configurations, in order to collect the set of clauses generated for each configuration. The simulator program then uses the corresponding set of generated clauses to run the positive and negative examples defined for each data set against them. To evaluate clauses, we used two different strategies: ***Pred*** denotes the tabling of individual predicates and ***Conj*** denotes the tabling of literal conjunctions (as described in [36]). By tabling conjunctions, we only need to compute them once. The strategy is then recursively applied as the ILP system generates more specific clauses, but this can increase the table memory usage arbitrarily.

Tables 6.6 and 6.5 show the table memory usage (columns ***Memory***), in MBytes, and the running times, in seconds, to store (columns ***Store***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Comp***) compiled tries for YapTab using the original table organization (column ***YapTab***), using the GT-T approach (column ***GT-T/YapTab***) and using the GT-ST design (column ***GT-ST/YapTab***). Again, for the GT-T and GT-ST approaches we only show the memory and running time ratios over YapTab's original table organization.

In general, the results in Table 6.6 confirm the results obtained in Table 6.4 for memory usage with both GT-T and GT-ST designs showing equivalent memory usage ratios. In particular, for the ***Pred*** strategy, memory usage showed to be, on average, 2% less for the GT-ST

| *Data Sets* | *YapTab* | | | |
|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* |
| ***Pred*** | | | | |
| **Carc_v1** | 1,669.0 | 68,524 | 72,088 | 84,658 |
| **Carc_v2** | 2.1 | 50,151 | 54,391 | 68,832 |
| **Muta_v1** | 0.6 | 96,578 | 5,072 | 5,456 |
| **Muta_v2** | 0.6 | 95,181 | 2,109 | 2,604 |
| ***Conj*** | | | | |
| **Carc_v1** | 18.5 | 652 | 588 | 536 |
| **Carc_v2** | a.m. | a.m. | a.m. | a.m. |
| **Muta_v1** | 84.8 | 102,214 | 6,792 | 7,309 |
| **Muta_v2** | 675.6 | 95,846 | 1,724 | 2,152 |

Table 6.5: Table memory usage (in MBytes) and store/load times (in seconds) for the ICLP benchmarks using YapTab's original table design.

| *Data Sets* | *GT-T/YapTab* | | | | *GT-ST/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|
| | *Memory* | *Store* | *Load* | *Comp* | *Memory* | *Store* | *Load* | *Comp* |
| ***Pred*** | | | | | | | | |
| **Carc_v1** | **0.62** | 1.15 | 1.13 | 1.12 | **0.60** | 1.07 | 1.02 | **0.97** |
| **Carc_v2** | **0.53** | 1.00 | 1.06 | 1.02 | **0.53** | 1.04 | 1.16 | 1.09 |
| **Muta_v1** | **0.62** | 1.09 | 1.07 | 1.04 | **0.60** | 1.06 | 1.08 | 1.06 |
| **Muta_v2** | **0.62** | **0.99** | 1.05 | 1.01 | **0.60** | 1.00 | 1.29 | 1.29 |
| ***Average*** | **0.59** | **1.09** | **1.08** | **1.06** | **0.57** | **1.11** | **1.18** | **1.18** |
| ***Conj*** | | | | | | | | |
| **Carc_v1** | **0.39** | **0.97** | **0.97** | 1.00 | **0.39** | | 1.04 | 1.10 |
| **Carc_v2** | - | - | - | - | - | - | - | - |
| **Muta_v1** | **0.53** | 1.00 | 1.06 | 1.02 | **0.53** | 1.04 | 1.16 | 1.09 |
| **Muta_v2** | **0.16** | 1.07 | **0.86** | **0.57** | **0.16** | 1.04 | **0.95** | **0.71** |
| ***Average*** | **0.36** | **1.01** | **0.96** | **0.86** | **0.36** | **1.05** | **1.05** | **0.96** |

Table 6.6: Table memory usage (in MBytes) and store/load times (in seconds) for the ILP benchmarks using YapTab with the support for the common global trie data structure.

design than GT-T. Since the ***Pred*** strategy tables individual predicates, the existence of complex compound terms reduces the memory spend when using GT-ST, although, these gains are residual. For the ***Conj*** strategy, both designs outperform the YapTab standard table organization. This happens because after a certain time, the ***Conj*** strategy will not table new terms, but only answers that are combinations of previous terms, therefore making the GT approach more feasible as it can share the representation of common terms appearing at different argument or substitution positions.

Regarding running time, the results in Table 6.6 also confirm and reinforce the results

obtained in Table 6.4. GT-T support outperforms the GT-ST design for storing and loading times and, for some configurations, it also outperforms the original YapTab design. This is the case for configurations either with or without compiled tries.

Finally, in Table 6.7, we present a new set of tests specially designed to provide more expressive results regarding the comparison between the GT-ST and the GT-T designs. In this tests, we have defined a tabled predicate `t/1` that simply stores in the table space terms defined by `term/1` facts and then we used a `test/0` predicate to call `t/1` with a free variable. We experimented `test/0` predicate with 9 different sets of 500,000 term facts of compound terms (with arity 1, 2, 3) where its arguments were also compound subterms (with arity 1, 3, 5). An example of such code for a compound term `f` with arity 2 containing arguments subterms with arity 3 (500,000 terms in total) is shown next.

```
:- table t/1.
t(A) :- term(A).

test :- t(A), fail.
test.

term(f(g(1,1,1), g(1,1,1))).
term(f(g(2,2,2), g(2,2,2))).
term(f(g(3,3,3), g(3,3,3))).
...
term(f(g(499998,499998,499998), g(499998,499998,499998))).
term(f(g(499999,499999,499999), g(499999,499999,499999))).
term(f(g(500000,500000,500000), g(500000,500000,500000))).
```

Opposed to the previous experiments, here we just used one free variable for the tabled predicate `t/1`. This difference is necessary, because when we have more than one free variable and, we produce different combinations between those free variables, we are raising the number of nodes represented in the local tries. More precisely, different combinations of free variables raises the number of answers and therefore the number of nodes in the local answer tries. Moreover, since these experiments serve the purpose to show the differences between the GT-T and GT-ST at memory level, we did not include the YapTab original table design in these experiments.

Table 6.7 shows the table memory usage (columns ***Tab.Memory***) composed by two columns one for total memory (columns ***Total***) and the other for GT's memory (columns ***GT***), in MBytes, and the running times, in milliseconds, to store (columns ***Store***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Comp***) compiled tries using the GT-T table design (column ***GT-T***), and using the GT-ST design (column ***GT-ST/GT-T***). For the values referring the GT-ST we only show the memory and running times ratios over the GT-T design. The running times are the average of five runs.

Table 6.7 suggests that the GT-ST outperforms the GT-T design in some special cases, the

| *500,000 Terms* | *GT-T* | | | | | *GT-ST/GT-T* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *Tab.Memory* | | | | | *Tab.Memory* | | | | |
| | *Total* | *GT* | *Store* | *Load* | *Comp* | *Total* | *GT* | *Store* | *Load* | *Comp* |
| *f/1* | | | | | | | | | | |
| **g/1** | 17.17 | 7.63 | 126 | 28 | 51 | 1.44 | 2.00 | 1.55 | 1.14 | 1.00 |
| **g/3** | 32.43 | 22.89 | 198 | 34 | 61 | 1.24 | 1.33 | 3.29 | 1.12 | 1.25 |
| **g/5** | 47.68 | 38.15 | 293 | 47 | 83 | 1.16 | 1.2 | 1.46 | 1.00 | **0.99** |
| *f/2* | | | | | | | | | | |
| **g/1** | 32.43 | 22.89 | 203 | 38 | 71 | 1.00 | 1.00 | 1.28 | 1.13 | 1.09 |
| **g/3** | 62.94 | 53.41 | 45 | 60 | 103 | **0.76** | **0.71** | 1.18 | **0.84** | **0.95** |
| **g/5** | 93.46 | 83.92 | 438 | 111 | 146 | **0.67** | **0.64** | 1.10 | **0.67** | **0.8** |
| *f/3* | | | | | | | | | | |
| **g/1** | 47.68 | 38.15 | 296 | 50 | 89 | **0.84** | **0.80** | 2.87 | 1.02 | 1.03 |
| **g/3** | 93.46 | 83.92 | 616 | 142 | 164 | **0.59** | 0.55 | 1.25 | **0.8** | **0.85** |
| **g/5** | 139.24 | 129.7 | 832 | 197 | 224 | **0.51** | **0.47** | **0.96** | **0.67** | **0.74** |
| *Average* | | | | | | **0.96** | **0.97** | **0.93** | **0.97** | **0.91** |

Table 6.7: Table memory usage (in MBytes) and store/load times (in seconds) for subterm representation using YapTab with support for the common global trie data structure.

results show three different situations, that can be distinguished by the arity of the functor term ***f***. For ***f/1*** terms, it clearly shows that the costs are higher for GT-ST, since it needs to store one extra node for every distinct subterm representation and there is no redundancy in the subterms. We can also see that the memory cost seems to be reduced when the subterm's arity increases from **g/1** to **g/5**. This occurs because the cost of the extra node for each subterm is diluted in the number of nodes represented in the GT.

The results on table 6.7 also show that, in some cases, the storing process can be a very expensive procedure. Remember that with the GT-ST support, we pay the cost of recreating the local tries/global trie interactions when checking/storing/loading a term inside the GT. A particular situation occurs for the case of ***f/2*** with subterms **g/1** where the memory spend is the same for both designs. This happens because the extra node used by GT-ST, to represent the reference to the subterm representation, is balanced by the arity of the functor term ***f***. From this point on, the GT-ST always outperforms the GT-T, not only for the system's memory, but also for the running times with and without compiled tries. These results suggest that, at least for some class of applications, GT-ST support has potential to achieve significant reductions on memory usage without compromising running time.

# Chapter 7

# Conclusions and Further Work

In this final chapter, we summarize the achievements of the work presented in this thesis, providing our conclusions and some directions for further work.

We have presented a new and more compact representation of list terms for tabled data that avoids the recursive nature of the WAM representation by removing unnecessary intermediate pair tokens. Our presentation followed the different approaches that we have considered until reaching our current final design. We focused our discussion on a concrete implementation, the YapTab system, but our proposals can be easy generalized and applied to other tabling systems. Our experimental results are quite interesting, they clearly show that with compact lists, it is possible not only to reduce the memory usage overhead, but also the running time of the execution for storing and loading list terms, both with and without compiled tries.

We also have presented three new designs for the table space organization, that have the common feature of representing all tabled subgoals and tabled answers only once in a common global trie instead of being spread over several different trie data structures. The goal of the GT designs starts by reducing to a minimum the nodes present in the subgoal and answer tries by moving the respective representation to the GT. Continues in the reduction of the redundancy in term representation by maximizing the sharing of tabled data that is structurally equal. And ends in the reduction of the redundancy of subterm representation in compound terms also maximizing the sharing of tabled data. Our experiments using the YapTab tabling system showed that our approaches have potential to achieve significant reductions on memory usage without compromising running time.

Further work will include, for the list term representation, exploring the impact of our proposal in real-world applications, such as, the works on Inductive Logic Programming and Probabilistic Logic Learning with the ProbLog language [37], that heavily uses list terms to represent, respectively, hypotheses and proofs in trie data structures. For the GT designs, further work include, exploring the impact of applying our proposal to other real-world

applications, that pose many subgoal queries, possibly with a large number of redundant answers, seeking real-world experimental results allowing us to improve and expand the current implementations.

# Appendix A

# Experimental Results for GT-CA

This appendix contains the results for the tabled predicate `t/5` and the ILP benchmark tests used in Chapter 6, obtained and published in [33] for a preliminary implementation of the GT-CA design. The environment for these experiments was an Intel(R) Core(TM)2 Quad 2.66GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24.23 with YapTab 5.1.4.

| *Terms* | *YapTab* | | | | *GT-CA/YapTab* | | | | *GT-T/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Mem* | *Str* | *Load* | *Cmp* | *Mem* | *Str* | *Load* | *Cmp* | *Mem* | *Str* | *Load* | *Cmp* |
| **1000 ints** | 191 | 1009 | 358 | 207 | 1.08 | 1.56 | 1.30 | n.a. | 1.00 | 1.32 | 1.18 | 1.69 |
| **1000 atoms** | 191 | 1040 | 337 | 231 | 1.08 | 1.54 | 1.41 | n.a. | 1.00 | 1.26 | 1.24 | 1.54 |
| **1000 f/1** | 191 | 1474 | 548 | 239 | 1.08 | 1.35 | 1.33 | n.a. | 1.00 | 1.28 | 1.11 | 1.88 |
| **1000 f/2** | 382 | 1840 | 632 | 353 | **0.58** | 1.25 | 1.37 | n.a. | **0.50** | 1.11 | 1.18 | 1.58 |
| **1000 f/4** | 764 | 2581 | 786 | 631 | **0.33** | 1.21 | 1.35 | n.a. | **0.25** | 1.07 | 1.16 | 1.14 |
| **1000 f/6** | 1146 | 3379 | 1032 | 765 | **0.25** | 1.12 | 1.29 | n.a. | **0.17** | 1.01 | 1.05 | 1.08 |
| **1000 [ ]/1** | 382 | 1727 | 466 | 365 | **0.58** | 1.32 | 1.44 | n.a. | **0.50** | 1.17 | 1.21 | 1.29 |
| **1000 [ ]/2** | 764 | 2663 | 648 | 459 | **0.33** | 1.06 | 1.55 | n.a. | **0.25** | **0.93** | 1.20 | 1.48 |
| **1000 [ ]/4** | 1528 | 4461 | 1064 | 720 | **0.20** | 1.10 | 1.57 | n.a. | **0.13** | **0.81** | 1.01 | 1.28 |
| **1000 [ ]/6** | 2293 | 6439 | 2386 | 1636 | **0.16** | 1.02 | 1.05 | n.a. | **0.08** | **0.71** | **0.58** | **0.68** |
| *Average* | | | | | **0.57** | **1.25** | **1.37** | **n.a.** | **0.49** | **1.07** | **1.09** | **1.36** |

Table A.1: Table memory usage (in MBytes) and store/load times (in milliseconds) for the `test/0` predicate using YapTab with and without support for the common global trie data structure.

| *Data Sets* | *YapTab* | | | | *GT-CA/YapTab* | | | | *GT-T/YapTab* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Mem* | *Str* | *Load* | *Cmp* | *Mem* | *Str* | *Load* | *Cmp* | *Mem* | *Str* | *Load* | *Cmp* |
| *Pred* | | | | | | | | | | | | |
| **Carc_P1** | 1.6 | 70.72 | 71.26 | 72.95 | **0.82** | 1.35 | 1.34 | n.a. | **0.62** | 1.07 | 1.05 | 1.03 |
| **Carc_P2** | 2.1 | 51.19 | 50.44 | 55.97 | **0.87** | 1.42 | 1.44 | n.a. | **0.51** | 1.23 | 1.30 | 1.22 |
| **Muta_P1** | 0.6 | 98.93 | 5.57 | 5.86 | **0.73** | 1.20 | 1.19 | n.a. | **0.63** | **0.91** | 1.00 | **0.94** |
| **Muta_P2** | 0.6 | 93.01 | 2.01 | 2.40 | **0.73** | 1.26 | 1.47 | n.a. | **0.63** | **0.96** | 1.22 | 1.10 |
| *Average* | | | | | **0.79** | **1.31** | **1.36** | **n.a.** | **0.60** | **1.04** | **1.14** | **1.07** |
| *Conj* | | | | | | | | | | | | |
| **Carc_C1** | 18.5 | 0.56 | 0.51 | 0.48 | **0.53** | 1.57 | 1.63 | n.a. | **0.39** | 1.20 | 1.22 | 1.08 |
| **Carc_C2** | 2802.8 | 93.85 | 70.16 | 36.44 | **0.50** | 1.50 | 1.50 | n.a. | **0.14** | 1.11 | 1.09 | **0.82** |
| **Muta_C1** | 84.7 | 97.02 | 7.36 | 6.14 | **0.66** | 1.30 | 1.65 | n.a. | **0.53** | **0.99** | 1.22 | 1.35 |
| **Muta_C2** | 675.6 | 92.76 | 1.36 | 1.53 | **0.16** | 1.25 | 1.42 | n.a. | **0.16** | **0.98** | 1.10 | **0.78** |
| *Average* | | | | | **0.46** | **1.41** | **1.55** | **n.a.** | **0.31** | **1.07** | **1.16** | **1.01** |

Table A.2: Table memory usage (in MBytes) and store/load times (in seconds) for the ICLP benchmarks using YapTab with and without support for the common global trie data structure.

# Bibliography


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

[2] P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, PhD thesis, University of California at Berkeley, 1990.

[3] R. Kowalski, "Predicate Logic as a Programming Language", in *Information Processing*. 1974, pp. 569–574, North-Holland.

[4] K. Apt and M. van Emden, "Contributions to the Theory of Logic Programming", *Journal of the ACM*, vol. 29, no. 3, pp. 841–862, 1982.

[5] H. Tamaki and T. Sato, "OLDT Resolution with Tabulation", in *International Conference on Logic Programming*. 1986, number 225 in LNCS, pp. 84–98, Springer-Verlag.

[6] W. Chen and D. S. Warren, "Tabled Evaluation with Delaying for General Logic Programs", *Journal of the ACM*, vol. 43, no. 1, pp. 20–74, 1996.

[7] K. Sagonas and T. Swift, "An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs", *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 3, pp. 586–634, 1998.

[8] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren, "Efficient Access Mechanisms for Tabled Logic Programs", *Journal of Logic Programming*, vol. 38, no. 1, pp. 31–54, 1999.

[9] R. Rocha, "On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation", in *International Symposium on Practical Aspects of Declarative Languages*. 2007, number 4354 in LNCS, pp. 155–169, Springer-Verlag.

[10] P. Costa, R. Rocha, and M. Ferreira, "Tabling Logic Programs in a Database", in *Workshop on (Constraint) Logic Programming*, 2007, pp. 125–135.

[11] H. Aït-Kaci, *Warren's Abstract Machine – A Tutorial Reconstruction*, The MIT Press, 1991.

[12] E. Goto, "Monocopy and Associative Algorithms in Extended Lisp", Tech. Rep. TR 74-03, University of Tokyo, 1974.

[13] J. Raimundo and R. Rocha, "A Very Compact and Efficient Representation of List Terms for Tabled Logic Program", in *Local Proceedings of the International Conference on Applications of Declarative Programming and Knowledge Management, INAP'2009*, S. Abreu and D. Seipel, Eds., Évora, Portugal, November 2009, pp. 157–170.

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

[15] R. Rocha, F. Silva, and V. Santos Costa, "On applying or-parallelism and tabling to logic programs", *Theory and Practice of Logic Programming*, vol. 5, no. 1 & 2, pp. 161–205, 2005.

[16] J. A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, 1965.

[17] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel, "Un système de communication homme–machine en francais", Technical report cri 72-18, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1973.

[18] D. H. D. Warren, *Applied Logic – Its Use and Implementation as a Programming Tool*, PhD thesis, Edinburgh University, 1977.

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

[20] W. Clocksin and C. Mellish, *Programming in Prolog*, Springer-Verlag, fourth edition, 1994.

[21] L. Sterling and E. Shapiro, *The Art of Prolog*, The MIT Press, 1994.

[22] K. Sagonas, D. S. Warren, T. Swift, P. Rao, S. Dawson, J. Freire, E. Johnson, B. Cui, M. Kifer, B. Demoen, and L. F. Castro, *XSB Programmers' Manual*, Available from `http://xsb.sourceforge.net`.

[23] M. Carlsson and J. Widen, "SICStus Prolog User's Manual", SICS Research Report R88007B, Swedish Institute of Computer Science, 1988.

[24] D. Michie, "Memo Functions and Machine Learning", *Nature*, vol. 218, pp. 19–22, 1968.

[25] R. Rocha, F. Silva, and V. Santos Costa, "A Tabling Engine for the Yap Prolog System", in *APPIA-GULP-PRODE Joint Conference on Declarative Programming*, 2000.

[26] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You, "Implementation of a Linear Tabling Mechanism", in *Practical Aspects of Declarative Languages*. 2000, number 1753 in LNCS, pp. 109–123, Springer-Verlag.

[27] Hai-Feng Guo and G. Gupta, "A Simple Scheme for Implementing Tabling based on Dynamic Reordering of Alternatives", in *Conference on Tabulation in Parsing and Deduction*, 2000, pp. 141–154.

[28] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren, "Efficient Tabling Mechanisms for Logic Programs", in *International Conference on Logic Programming*. 1995, pp. 687–711, The MIT Press.

[29] L. Bachmair, T. Chen, and I. V. Ramakrishnan, "Associative Commutative Discrimination Nets", in *International Joint Conference on Theory and Practice of Software Development*. 1993, number 668 in LNCS, pp. 61–74, Springer-Verlag.

[30] J. Raimundo and R. Rocha, "Compact Lists for Tabled Evaluation", in *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages, PADL'2010*, M. Carro and R. Peña, Eds., Madrid, Spain, January 2010, number 5937 in LNCS, pp. 249–263, Springer-Verlag.

[31] J. Costa and R. Rocha, "Global Storing Mechanisms for Tabled Evaluation", in *Proceedings of the 24th International Conference on Logic Programming, ICLP'2008*, M. Garcia de la Banda and E. Pontelli, Eds., Udine, Italy, December 2008, number 5366 in LNCS, pp. 708–712, Springer-Verlag.

[32] J. Costa and R. Rocha, "One Table Fits All", in *Proceedings of the 11th International Symposium on Practical Aspects of Declarative Languages, PADL'2009*, A. Gill and T. Swift, Eds., Savannah, Georgia, USA, January 2009, number 5418 in LNCS, pp. 195–208, Springer-Verlag.

[33] J. Costa, J. Raimundo, and R. Rocha, "A Term-Based Global Trie for Tabled Logic Programs", in *Proceedings of the 25th International Conference on Logic Programming, ICLP'2009*, P. Hill and D. S. Warren, Eds., Pasadena, California, USA, July 2009, number 5649 in LNCS, pp. 205–219, Springer-Verlag.

[34] S. Muggleton, "Inductive Logic Programming", in *Conference on Algorithmic Learning Theory*. 1990, pp. 43–62, Ohmsma.

[35] N. A. Fonseca, F. Silva, and R. Camacho, "April - An Inductive Logic Programming System", in *European Conference on Logics in Artificial Intelligence*. 2006, number 4160 in LNAI, pp. 481–484, Springer-Verlag.

[36] R. Rocha, Nuno A. Fonseca, and V. Santos Costa, "On Applying Tabling to Inductive Logic Programming", in *European Conference on Machine Learning*. 2005, number 3720 in LNAI, pp. 707–714, Springer-Verlag.

[37] A. Kimmig, V. Santos Costa, R. Rocha, B. Demoen, and L. De Raedt, "On the Efficient Execution of ProbLog Programs", in *Proceedings of the 24th International Conference on Logic Programming, ICLP'2008*, M. Garcia de la Banda and E. Pontelli, Eds., Udine, Italy, December 2008, number 5366 in LNCS, pp. 175–189, Springer-Verlag.