Miguel João Gonçalves Areias

On Applying Linear Tabling to Logic Programs

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

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

Miguel João Gonçalves Areias

On Applying Linear Tabling to Logic Programs

Dissertação submetida à Faculdade de Ciências da Universidade do Porto como parte dos requisitos para a 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
Setembro de 2010

Dedicated to

Rita, my Parents and Bubba

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Ricardo Rocha, whose encouragement, guidance and support shaped this research. His guidance helped me, all the time, during the research and writing of the thesis. I could not have imagined having a better advisor and mentor for my MSc study.

Besides my advisor, I would like to acknowledge Prof. Luís Lopes, Prof. Víctor Santos Costa, Prof. Inês Dutra and Prof. Fernando Silva, for their encouragement and insightful comments made during my Msc course.

To all my fellow colleagues from the STAMPA project, João Santos, João Raimundo, José Vieira and Flávio Cruz for their support and excellent work environment. Wish you all the best for your professional and personal future.

To Jorge Torres, Sílvio Almeida and all friends, Professors and assistances from the DCC-FCUP for creating a very good environment to study Computer Science.

Finally, I would like to thank my family and friends for all the support during the last years, specially to Rita Pires and to my parents, João Areias and Margarida Areias that believed, trusted and supported me even more than I could ever imagine. A peace of each one of you is present in this work. Thank you all.

September 2010

Miguel Areias

Abstract

Logic programming languages, such as Prolog, are derived from Horn Clause Logic and provide a well understood resolution based inference mechanism. Although Prolog is a popular and successful language, its potential is limited by the SLD resolution method on which it is based. SLD resolution was proven to be inefficient when dealing with infinite loops and redundant subcomputations. Tabled evaluation is a recognized and powerful technique that overcomes those limitations on traditional Prolog systems based on SLD resolution. We can distinguish two main categories of tabling mechanisms: suspension-based tabling and linear-based tabling. While suspension-based mechanisms are considered to obtain better results in general, they have more memory space requirements and are more complex and hard to implement than linear tabling mechanisms.

The work presented on this thesis was focused on making a deep study about linear tabling, in order to understand how different linear tabling strategies can affect the evaluation flow of tabled programs and improve its overall performance. Arguably, the SLDT and DRA strategies are the two most successful extensions to standard linear tabled evaluation. In this work, we propose a new strategy, named DRS, and we present a framework, on top of the Yap system, that supports the combination of all these three linear tabling strategies. Our implementation shares the underlying execution environment and most of the data structures used to implement tabling in the YapTab engine, which is the actual suspension-based tabling mechanism of the Yap Prolog system. All these common features allows us to make a first and fair comparison between the linear tabling strategies, used solely or combined with the other, and YapTab's suspension-based mechanism, in order to better understand the advantages and weaknesses of each feature. The obtained results confirmed that suspension-based mechanisms have, in general, better performance than linear tabling and that the difference between both mechanisms can be highly reduced by using the correct combination of linear tabling strategies.

Resumo

As linguagens de Programação em Lógica que derivam da lógica de Horn, tal como o Prolog, têm mecanismos de resolução baseados em inferência que são bastante conhecidos. Embora o Prolog seja uma linguagem com bastante sucesso, o seu potencial é limitado pelo seu mecanismo de resolução, que é baseado na resolução SLD. O mecanismo de resolução SLD foi provado ser bastante ineficiente quando avalia programas lógicos que têm ciclos infinitos ou sub-computações redundantes. A tabulação é uma técnica de implementação bastante reconhecida e poderosa que permite ultrapassar essas limitações em sistemas de Prolog que são baseados na resolução SLD. Actualmente, a técnica de tabulação pode ser dividida em dois grandes mecanismos: por suspensão das pilhas de execução e por execução linear. Os mecanismos por suspensão das pilhas de execução são considerados terem melhores resultados, no entanto eles têm mais requisitos em termos de memória e são mais complexos de implementar do que os mecanismos lineares.

O trabalho apresentado nesta tese pretende fazer um estudo aprofundado sobre os mecanismos de tabulação linear, de forma a perceber como as diferentes estratégias de tabulação afectam o fluxo de avaliação de um programa lógico e melhoram a performance geral do sistema. As estratégias SLDT e DRA são duas das mais conhecidas e bem sucedidas estratégias implementadas em sistemas de tabulação linear. Neste trabalho, propomos uma nova estratégia, que foi denominada de DRS, e apresentamos uma plataforma integrada, que suporta a combinação das três estratégias. A nossa implementação partilha o ambiente de execução e a maioria das estructuras de dados usadas pela máquina de execução do YapTab, que é o actual mecanismo de tabulação baseado em suspensão de pilhas do sistema Yap Prolog. A combinação de todas as estratégias e mecanismos na nossa plataforma permitiu-nos fazer uma primeira comparação justa entre todas as estratégias lineares, usadas sozinhas ou combinadas, e o mecanismo original do YapTab, de forma a perceber as vantagens e desvantagens de cada um. Os resultados obtidos, confirmam que os mecanismos baseados em suspensão

têm, no geral, melhores resultados do que os mecanismos lineares, sendo que a diferença entre os resultados de ambos os sistemas pode ser em grande parte reduzida através da combinação correcta das melhores estratégias lineares.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The main goal of a programming language is to enable the communication between humans and machines in order to define problems and their general means to obtain solutions. The first programing languages were machine languages. To communicate, the programmer had to learn the psychology of the machine and to express problems in machine-oriented terms. Higher-level languages, developed from machine languages, through the provision of facilities for the expression of problems in terms closer to their original conceptualization. It is believed that higher-level languages are particularly helpful in developing succinct and correct programs that are easy to write and understand. Logic programming languages, together with functional programming languages, form a major class of languages, called *declarative languages*, and because they are based on the predicate calculus, they have a strong mathematical base. Arguably, Prolog is the most popular and powerful logic programming language. Prolog gained its popularity mostly because of the success of the sophisticated compilation technique and abstract machine known as the WAM (Warren's Abstract Machine), presented by David H.D. Warren in 1983 [36].

A Prolog program is a set of clauses (logical sentences) written in a subset of first-order logic called *Horn clause logic*, that can be interpreted as *if-statements*. A *predicate* is a set of clauses that defines a relation, i.e., all the clauses have the same name and arity (number of arguments). Predicates are often referred by the pair *name/arity*.

The operational semantics of Prolog is given by SLD resolution, an evaluation strategy particularly simple that matches current stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. Tabling is a recognized and powerful implementation technique that overcomes the limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion and that can considerably reduce the search space, avoid looping and have better termination properties than SLD resolution [5].

Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [29], the most successful engine of XSB. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Implementations of tabling are now widely available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog. In these implementations, we can distinguish two main categories of tabling mechanisms: *suspension-based tabling* and *linear tabling*.

Suspension-based tabling mechanisms need to preserve the computation state of suspended tabled subgoals in order to ensure that all answers are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. The SLG-WAM [29] and the YapTab model [28] preserve the environment of a suspended computation by freezing the stacks. The Mercury implementation [32] and two alternative XSB-based models, the CAT [9] and the CHAT [10] models, copy the execution stacks to separate storage. Two more recent approaches, implemented in Yap [27] and Ciao Prolog [6], feature a higher-level implementation of suspension-based tabling. They apply source level transformations to a tabled program and then use external tabling primitives to provide direct control over the search strategy. In these proposals, suspension is implemented by leaving a *continuation call* [23] for the current computation in the table entry corresponding to the repeated call being suspended.

On the other hand, linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The main idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. Two different linear tabling proposals are the SLDT strategy of Zhou *et al.* [39], as originally implemented in B-Prolog, and the DRA technique of Guo and Gupta [12], as originally implemented in ALS-Prolog. The key idea of the SLDT strategy is to let repeated calls execute from the backtracking point of the former call. The repeated call is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a fix-point is reached. The DRA technique is based on dynamic reordering of alternatives with repeated calls. This technique tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls. It then uses those alternatives to repeatedly recompute them until reaching a fix-point.

There are two major scheduling strategies that can be used for both tabling mechanisms: *local scheduling* and *batched scheduling* [11]. The local scheduling strategy allows a cluster of subgoals to return answers only after the fix-point has been reached. The batched scheduling strategy schedules the program clauses in a depth-first manner as does the WAM. It favors forward execution first, backtracking next, and consuming answers or completion in last.

1.1 Thesis Purpose

With this thesis, we intended to create a common framework to support both linear-based and suspension-based tabling mechanisms for the local and batched scheduling strategies, in order to analyze the advantages and weaknesses of each mechanism. Our framework shares the underlying execution environment and most of the data structures already available to implement suspension-based tabling in Yap, known as the YapTab engine [28]. In particular, we took advantage of YapTab's efficient table space data structures based on *tries* [22], which we used with minimal modifications.

Accordingly, our goal was to extend the current YapTab engine with new structures and instructions that would allow the efficient implementation of linear tabling mechanisms. We begin the implementation, with the conceptual design of a standard linear tabling system, and only afterwards we proceed with support for optimizations. For that, we studied the optimizations already implemented on other linear tabling systems, in order to select the ones to be implemented on our system and, as consequence, we also propose a new strategy for the local scheduling strategy, which we called *Dynamic Reordering of Solutions (DRS)*.

Arguably, the SLDT [39] and DRA [12] strategies are the two most successful extensions to standard linear tabling evaluation. In our work, we present a new and efficient implementation of both strategies. The innovation will be to consider that both strategies schedule the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of YapTab [3].

As the SLDT, DRA and DRS strategies optimize different aspects of the evaluation, they are, in principle, orthogonal to each other and thus it should be possible to combine both in the same system. However, to the best of our knowledge, no single Prolog system supports all strategies simultaneously. Our intention is then to have a linear tabling framework that supports simultaneously all these optimizations, in order to allow us to make a first and fair comparison between these different linear tabling strategies and, therefore, better understand the advantages and weaknesses of each, when used solely or combined with the others.

In order to better understand the results of the linear tabling mechanism and its optimizations, we also created a engine's component to gather statistical information during the evaluation. This small component is only enabled when we are interested in taking statistical information, thus it will not affect the performance of the system during a normal evaluation.

1.2 Thesis Outline

This thesis is structured into seven chapters that reflect the work developed. Next, follows a summary of the main ideas presented and discussed in each chapter.

Chapter 1: Introduction. The current chapter.

Chapter 2: Logic Programming and Tabling. Introduces the basic concepts of Logic Programming and the Prolog language. The focus is them given to tabling techniques applied to Prolog systems.

Chapter 3: Linear Tabling Strategies. Describes the key concepts behind standard linear tabled evaluation and its optimizations when using the local scheduling strategy. The first section of this chapter presents a general evaluation that uses the standard linear approach and the remaining sections present how that evaluation changes accordingly to the different optimizations implemented on the linear tabling system.

Chapter 4: Implementation Details. This chapter describes the main low-level details involved in the implementation of the standard linear tabling engine and its optimizations for the local scheduling strategy. It also describes, how the different structures implemented on the system interact with each other.

Chapter 5: Batched Scheduling. Presents all the changes and low-level details necessary to extend the system to support the batched scheduling strategy.

Chapter 6: Performance Analysis. Analyzes the advantages and weaknesses of each optimization implemented on the linear tabling system, when used solely or combined with each other. It presents running time results and internal statistics taken during the evaluation of a set of benchmark tests. The chapter concludes with a comparison between our linear tabling system and YapTab's original suspension-based mechanism.

Chapter 7: Conclusions. Discusses the research and contributions made to the state-of-the-art linear tabling systems and suggests directions for future work.

In order to lighten the results presented on chapter 6 and make it more reader friendly, some details about the full results obtained were moved to the appendixes A and B. In chapter 6, the reader can find more details about this situation.

Chapter 2

Logic Programming and Tabling

In order to make this thesis as most self-contained as possible, this chapter introduces the basic concepts related with Logic Programming and the Prolog language. Since tabling is the center topic of this thesis, focus will then be given to the state-of-the-art tabling techniques for Prolog systems.

2.1 Logic Programming

Logic Programming roots were started mostly with Robinson in 1965, when he began the research for an automated theorem proving tool, on his work about the *Resolution Principle* [25]. The resolution principle is based on the induction principle "*if the implication $A \Rightarrow B$ is true, then to prove B, it is sufficient to prove A*". The expression *Logic Programming* was introduced afterwards by Kowalski, to designate the use of logic as the theoretical base for computer programming languages [14]. Kowalski showed how *SLD-resolution (Selected Linear Deduction)* treats implications as deduction procedures. Kowalski and Kuehner argued that SLD-resolution was the best inference system for first order logic, because it fills the following criteria [15]:

- Admits few redundant *deductions* and limits those which are irrelevant to a proof;
- Admits simple *proofs*;
- Determines a *search space* which is amenable to a variety of methods for heuristic search.

The completeness of SLD-resolution ensures that, by applying the SLD-resolution to a theory (or computer program) and a query, is it possible to use the theory to search for all solutions that satisfy the query [7, 2, 17].

Logic Programming is based on *predicate calculus.* An algorithm is seen as a set of two disjoint elements: *logic* and *control.* The logic component corresponds to the definition of the problem to be solved, while the control component, defines how the solution can be reached. The programmer needs only to *specify* the logic component of the algorithm, which is the problem to be solved, and leave the control of execution to the Logic Programming system.

According to Kalrsson [13], Logic Programming has the following major features:

- Variables are *logical* variables which can be instantiated *only once*;
- Variables are *untyped* until instantiated;
- Variables are instantiated via *unification*, a pattern matching operation finding the most general common instance of two data objects;
- At unification failure the execution *backtracks* and tries to find another way to satisfy the original query.

Common literature, also recognizes that Logic Programming has the following advantages [4]:

- **Simple declarative semantics**. A logic program is simply a collection of predicate logic clauses.
- **Simple procedural semantics**. A logic program can be read as a collection of recursive procedures. Clauses are tried in the order they are written and goals within a clause are executed from left to right.
- **High expressive power**. Logic programs can be seen as executable specifications that despite their simple procedural semantics allow for designing complex and efficient algorithms.
- **Inherent non-determinism**. Since in general several clauses can match a goal, problems involving search are easily programmed in these kind of languages.

These advantages lead to a more flexible programming style, in the sense that programs are more easy to understand, transform and/or expand.

The basic data structures for logic programs are called **terms**. Terms can be constants, variables or functors (functional terms). A **functor** can be identified by name and arity (number of arguments). For example, **f/n** denotes the functional term $f(t_1, ..., t_n)$, where t_1 to t_n are themselves terms and called the **arguments** of **f**. **Constants** can be considered functors with arity zero.

A **literal** is similar to a term, except that literals form individual goals to which a truth value can be assigned.

A **substitution** (or unification) is a finite set (possibly empty) of pairs with the form $X = t$, where X is a variable and t is a term, and there can be only one pair with X on the left side of the equality.

A logic program is a finite set of **clauses**. Each clause has the logic form:

$$\forall_{\vec{X}}(A \Leftarrow B_1 \wedge B_2 \wedge ... \wedge B_n)$$

where, A is called the **head**, $B_1 \wedge B_2 \wedge ... \wedge B_n$ is called the **body**, individual B_i's are called **goals** and $\vec{X}$ denotes the vector of variables present on the clause. If $n = 0$, the clause is called a **fact**.

A **computation** of a logic program corresponds to logically deduct goals from the substitution of a given query with program clauses. **Queries**, have the following logic form:

$$\forall_{\vec{X}}(\Leftarrow B_1 \wedge B_2 \wedge ... \wedge B_n)$$

where $\vec{X}$ denotes the vector of variables present on the query.

2.2 The Prolog Language

The Prolog language, which is one of the most popular Logic Programming languages, has its origins in a software tool implemented by Colmerauer in 1972 at the *Université de Aix-Marseille*, that was named *PROgramation en LOGic* [8].

Prolog is based on Horn clauses, which are defined as,

$$\forall n \geq 1(c(X) : -g_1(Y), ..., g_n(Z).)$$

for clauses with head and body, and

$$c(X).$$

for fact clauses. The symbol :- represents the implication $\Leftarrow$, the comma (,) represents the conjunction symbol $\wedge$. The X, Y and Z, represent the set (possibly empty) of terms of each goal.

There are several Prolog implementations and models of execution. Pure and sequential Prolog's execution consists in traversing a search tree in a *depth-first left-to-right* form, as shown in the example of Figure 2.1.

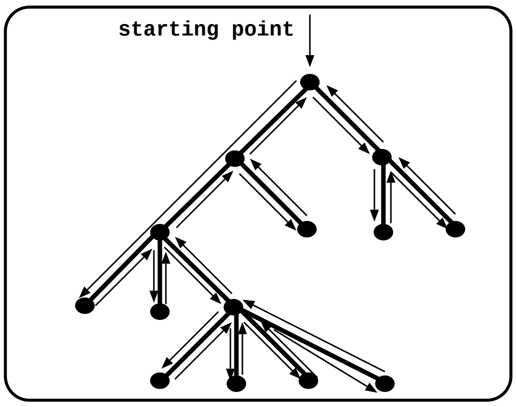


Figure 2.1: Depth-first search with backtracking in Prolog

Non leaf nodes of the search tree represent stages of computation (*choice points*) where alternative branches (clauses) can be explored, to satisfy program's query, while leaf nodes represent solution or fail nodes. When the computation reaches a non leaf node and can not advance any further, Prolog starts the *backtracking* mechanism, which consists in restoring the computation up to the previous node and schedule an alternative unexplored branch. A programmer can optimize the default search procedure by pruning the search tree through the use of the *cut* operator (!). Cut allows programs to use less memory and to be faster, because it reduces the allocation of backtracking nodes and the search space [35].

Some major characteristics of Prolog systems can be resumed as follows:

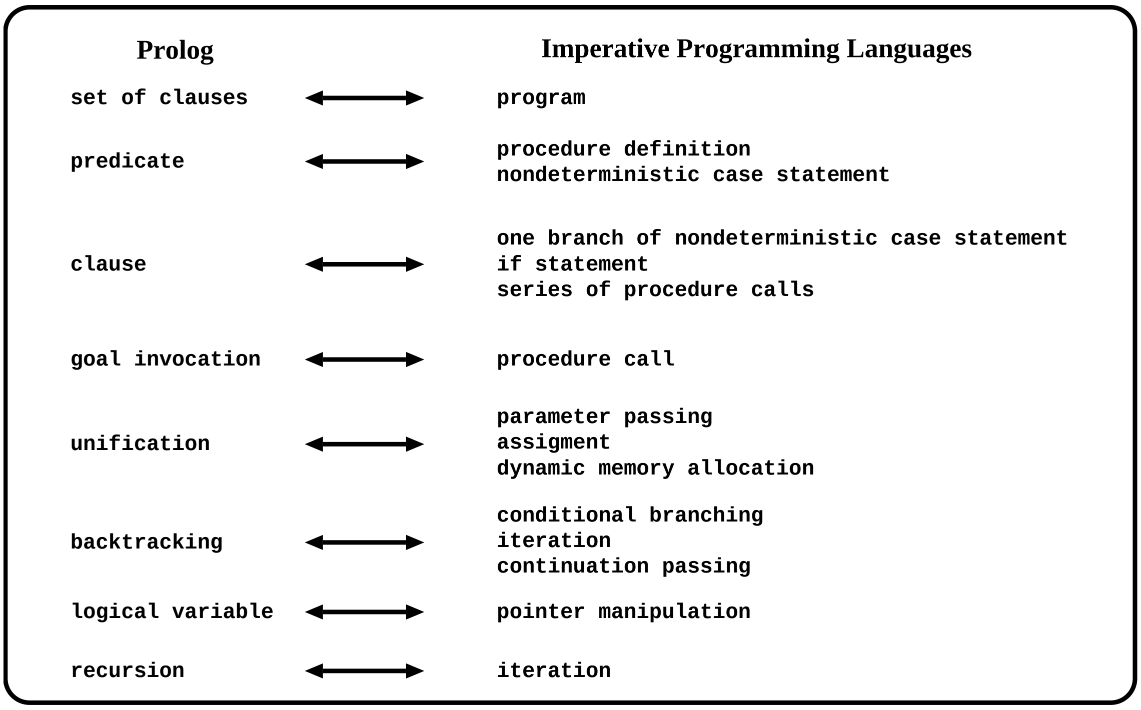
- It is a system oriented for symbolic processing.

- Presents a declarative semantic inherent to logic.
- Supports iterative and recursive programs.
- Represents programs and data with the same formalism.
- Allows different answers for the same query.

When comparing performance with imperative languages, Prolog's execution can be seem as a natural generalization of the execution of imperative languages. The Prolog language can be summarized as:

$$\text{Prolog} = \text{imperative language} + \text{unification} + \text{backtracking}$$

As in imperative languages, the execution flow is left to right within a clause. The goals in the body of a clause are called like procedures. When a goal is called, the program clauses which match with it, are chosen in the top-bottom textual order. Figure 2.2 resumes the relation between concepts used in Prolog and in imperative programming languages.



Prolog		Imperative Programming Languages
set of clauses	⟷	program
predicate	⟷	procedure definition nondeterministic case statement
clause	⟷	one branch of nondeterministic case statement if statement series of procedure calls
goal invocation	⟷	procedure call
unification	⟷	parameter passing assigment dynamic memory allocation
backtracking	⟷	conditional branching iteration continuation passing
logical variable	⟷	pointer manipulation
recursion	⟷	iteration

Figure 2.2: Correspondence between concepts used in Prolog and in imperative programming languages

In general, Prolog's performance is lower than imperative languages, due to the extra control and structures required by Logic Programming, but the trade-offs are good enough for a logical and efficient programming style to be possible [19].

2.3 The Warren's Abstract Machine

Most of the currently available Prolog systems are based on a sophisticated compilation technique and abstract machine known as the WAM (Warren's Abstract Machine). The WAM was originally proposed by David H. D. Warren [36, 37] and its compiler correctness was later formally verified by Pusch in the work [20].

The tutorial book on the WAM [1], describes the WAM as a sequence of engines that incrementally support the different functionalities of a pure Prolog system. This division in incremental engines, benefits the presentation and comprehension of all the small tasks involved in the complex problem which is the implementation of a Prolog system. The minimal engine is the abstract machine M_0, which is only capable of determining whether a goal unifies with a given term. The abstract machine M_1 extends M_0, by allowing programs with more than one fact and with at most one fact per predicate name. The machine M_2, which is the next stage, is capable of compiling Prolog with conjunction rules (that is, with the form $a_0 : -a_1, ..., a_n$). The machine M_3, allows disjunctive definitions (more that one rule for each predicate), by adding the *backtracking* mechanism. Finally, the complete Prolog system is reached, by adding support for cuts, constants, lists and anonymous variables. Different Prolog systems employ also various design optimizations, such as swapping final instructions and/or avoiding the allocation of environments in special cases. The main goal behind all these optimizations is to reduce the computation's execution time and/or use as less memory as possible.

At the implementation level, the WAM is defined by a set of data structures, a set of registers and a set of low-level instructions.

Regarding the memory organization of the WAM, it is divided in seven logical data structures: two stacks for data objects (the global and the local stacks), one stack to support the interaction between the unification and the backtracking mechanisms (the trail), one stack to support unification (the PDL), one stack for the code area, one stack for the table of symbols and one array to store argument and temporary registers.

- **Global stack** or **heap**. It is an array of data cells used to represent compound data terms, such as lists and structures.

- **Local stack**. It holds *environments* and *choice points*. Environments (also known as *local frames*) store the variables that are local to a clause and the continuation pointer. Choice points are used to store the current state of the computation. This means that, whenever a predicate starts execution, a choice point is allocated, with information of execution's state up to that moment, and with information about unexploited alternatives to be explored via the backtracking mechanism.

- **Trail**. It is used to store the addresses of the variables which must be unbound when backtracking occurs.

- **Push-Down List (PDL)**. This stack is used by the unification process when handling nested compound terms.

- **Code area**. This area contains the WAM compiled code of the programs loaded.

- **Symbol table**. Used to store information about the symbols, such as atoms or structures. An example is the mapping between the internal representation of a term and it's printing name.

- **Register's array**. Used to store the arguments of the calls made during the evaluation and temporary registers.

The registers used to control WAM's flow of execution are described in Fig. 2.3. The purpose of most registers is straightforward, but some can be not so obvious. For example, the HB register caches the value of H stored in the most recent choice point. The S register is used during unification of compound terms (terms with arguments) and points to the argument being unified. The arguments are accessed one by one by successively incrementing this pointer. Some instructions have different behaviors during read and write mode unification, and the mode flag is used to distinguish between both situations.

Figure 2.4 shows the correspondence between registers and stacks. It also shows the information stored by choice points, environments and data terms. The choice points store all key registers needed to restore the computation and launch the alternative program clauses, the backtracking registers used to put the computation on the previous stage and the arguments of the present call. The environments store the

P	**Program counter**	
CP	**Continuation Pointer**	**(top of return stack)**
E	**current Environment pointer**	**(in local stack)**
B	**most recent Backtrack point**	**(in local stack)**
A	**top of local stack**	**(max between E and B)**
TR	**top of TRail**	
H	**top of Heap**	
HB	**Heap Backtrack point**	**(in heap)**
S	**Structure pointer**	**(in heap)**
Mode	**Mode flag**	**(read or write)**
A1,A2,...	**Argument registers**	
Y1,Y2,...	**temporary registers**	

Figure 2.3: WAM's registers

current active choice point and the permanent variables, i.e., the variables that appear in more than a body subgoal, for the current alternative clause.

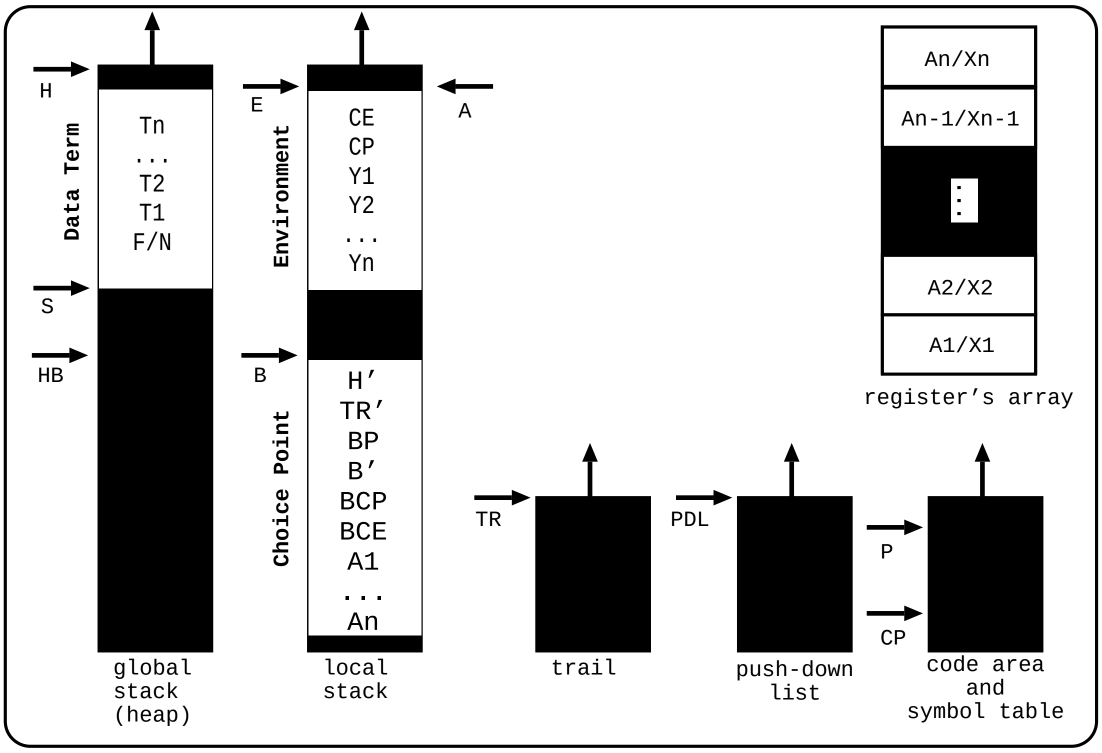


Figure 2.4: WAM's memory organization and registers

Regarding the low-level instruction set of the WAM, it can be divided into four major groups. The most relevant instructions per group to be noticed are:

- **Choice point instructions**. They allow the allocation/deallocation of choice points and the recovery of the computation state stored on those choice points.
  - **try_me_else** *L*: creates a choice point and sets L (label) as the next alternative for the choice point.
  - **retry_me_else** *L*: recovers the computation's state stored on the top most choice point and updates the next alternative for the choice point to be L.
  - **trust_me**: recovers the computation's state stored on the top most choice point and removes the top-most choice point from the local stack.
  - **try** *L*: creates a choice point, sets the next instruction as the next alternative for the choice point and moves the execution to L.
  - **retry** *L*: recovers the computation's state stored on the top most choice point, updates the next alternative for the choice point to be the next instruction and moves the execution to L.
  - **trust** *L*: recovers the computation's state stored on the top most choice point, removes the top-most choice point from the local stack and moves the execution to L.
- **Control instructions**. Used to allocate/remove environments and manage the call/return sequence of subgoals.
  - **allocate/deallocate**: used to create and remove environments, respectively.
  - **call** *pred,N*: calls the predicate *pred* and trims the current environment size to *N* (*N* represents the number of permanent variables that should be kept).
- **Unification instructions**. These instructions implement specialized versions of the unification algorithm according to the position and type of the arguments.
  - The **get_** instructions are used for head unification with registers.
  - The **unify_** instructions are used for head unification with structure arguments.
  - The **put_** and **set_** instructions are used for loading argument registers just before a call.
- **Indexing instructions**. These type of instructions accelerate the process of determining which clauses unify with a given predicate. Depending on the first

argument of the call, they jump to specialized code that can directly index the unifying clauses.

- The **switch_on_term** instruction is used to jump to specialized code accordingly to the type of term (being a variable, a constant, a list or a structure).
- The **switch_on_constant** instruction indexes the clauses which match with a constant term.
- The **switch_on_structure** instruction indexes the clauses which match with a structure term.

2.4 Tabling on Prolog Systems

The Prolog language is based on the combination of the SLD-resolution mechanism with linear top-down exploration of clauses defined in a program. This combination can be incomplete for certain types of programs. The cause of this incompleteness is the presence of recursive predicates during the evaluation of a program, because they can lead to the infinite exploration of the same search space.

One *well-known* logic program which can lead to this behavior is the *path problem.* Consider for example that, the predicate *edge/2* defines the transition function of a graph. The function $edge_1$, defines a direct acyclic graph with three nodes and the function $edge_2$, defines a direct cyclic graph with two nodes. The functions $path_1$ and $path_2$ are two equivalent forms of defining a generic path inside a graph.

$$edge_1 = \begin{cases} edge(1,2). \\ edge(2,3). \end{cases}$$

$$edge_2 = \begin{cases} edge(1,2). \\ edge(2,1). \end{cases}$$

$$path_1 = \begin{cases} path(X,Z) : -edge(X,Y), path(Y,Z). \\ path(X,Z) : -edge(X,Z). \end{cases}$$

$$path_2 = \begin{cases} path(X,Z) : -path(X,Y), edge(Y,Z). \\ path(X,Z) : -edge(X,Z). \end{cases}$$

Consider now that we would like to use both definitions of *path*, to evaluate on a standard Prolog system, all the nodes we can reach on both graphs, starting from node 1. We will use then, the query goal $path(1, Z)$ against $path_1$ and $path_2$. Both $path$ definitions are logically correct and equivalent, so it would be expectable to get successful evaluations and equal solutions, when the evaluating ends. The solution set of $edge_1$ graph is $\{Z = 2,Z = 3\}$ and the solution set of $edge_2$ graph is $\{Z = 1,Z = 2\}$.

Figure 2.5 shows the evaluation tree of $path_1$ with graph $edge_1$. The Prolog system uses the first clause from the path definition that matches the call and in the continuation calls the subgoal $path(2, Z)$. Then, it uses again the first matching clause and calls the subgoal $path(3, Z)$. This predicate don't have any matching clause, so the computation fails. This means, that the computation backtracks to the previous unexploited clause and the evaluation reaches to the answer $Z = 3$. Then, the Prolog system backtracks again to the previous unexploited clause, that matches subgoal $path(1, Z)$ and the evaluation reaches to the answer $Z = 2$. As there aren't any unexploited clauses, all the answers were found and the Prolog system finishes the evaluation. The result of the evaluation was correct, because it matched the solution set.

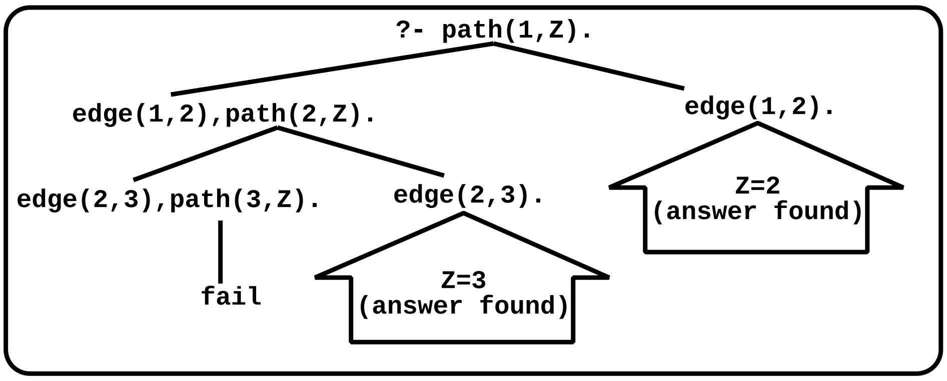


Figure 2.5: Evaluation of $path_1$ with $edge_1$

Figure 2.6(a) shows the evaluation tree of $path_1$ with $edge_2$. The Prolog system uses the first matching clause for the subgoal $path(1, Z)$, the evaluation reaches to the subgoal $path(2, Z)$, which calls again the subgoal $path(1, Z)$. This recursive call to $path(1, Z)$, defines a positive loop, but the Prolog system can not detect it, so the Prolog system will start another evaluation of $path(1, Z)$. This leads to the infinite repetition of the same sub-computation, so the Prolog system will not find any solution.

Figure 2.6(b) shows the evaluation tree of $path_2$ with $edge_1$ and $edge_2$. For both graphs, the Prolog system starts again, by using the first clause which matches the call $path(1, Z)$, but it leads to predicate $path(1, Y)$, which is a repeated call (also known as a variant)[1] to $path(1, Z)$. Again, a positive loop is found, so the Prolog system will not find any solution for this problem.

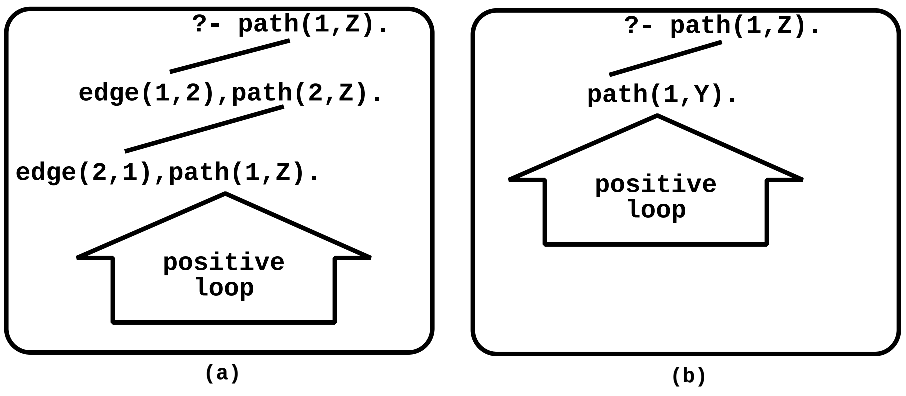


Figure 2.6: Evaluation of (a) $path_1$ with $edge_2$ and (b) $path_1$ with both $edge$ functions

Therefore, we have seen that, when a Prolog system does not find positive loops during the evaluation, it returns the correct solutions, but when it finds a positive loop, it evaluates the same sub-computation infinitely without reaching to any solution.

This raises two major problems for standard Prolog systems. The first is that theory ensures that the evaluation of Horn Clauses is complete, but the declarative advantage of logic programs became dependent on the programmer's capability of designing his programs with clauses in the correct order. The second is that Prolog systems can not be used in important applications, such as Deductive Databases.

The operational incompleteness of Prolog is a well known problem and several proposes to improve Prolog's declarativeness exist. Next we will discuss one of such proposals, generically known as *tabling* (also known as *tabulation* or *memoing*) [18].

2.4.1 General Idea

The key idea of tabling is to use an auxiliary data space, the *table space*, to keep track of the subgoal calls in evaluation and store, for each subgoal, the set of *answers* which are found during program's evaluation. Whenever a subgoal has a repeated call, the subgoal is resolved by consuming answers from table space instead of executing the program clauses. In the meantime, as new answers are found, they are added to their tables and later returned to all repeated calls. By using answer resolution in

[1]Variant calls of a subgoal are calls which differ only on variable renaming.

this manner instead of program resolution as usual, tabling based systems can avoid looping and reduce the search space of programs [5].

The OLDT was one of the first approaches used to supply the incompleteness of standard Prolog systems. It was presented by Tamaki and Sato, and combines the use of OLD resolution with a tabling technique [34]. The SLG resolution [5], implemented after OLDT, is another tabling mechanism that has been gaining popularity, since its implementation on the XSB Prolog system [24, 30].

The XSB design uses an adapted version of the standard WAM, called SLG-WAM [29], that extends SLD-resolution with new tabling related structures. The SLG-WAM defines nodes in a different way from the WAM. Nodes are defined as *generators* if they correspond to first calls of tabled subgoals, *consumers* if they correspond to repeated calls to tabled subgoals, and *interior* if they correspond to non-tabled subgoals.

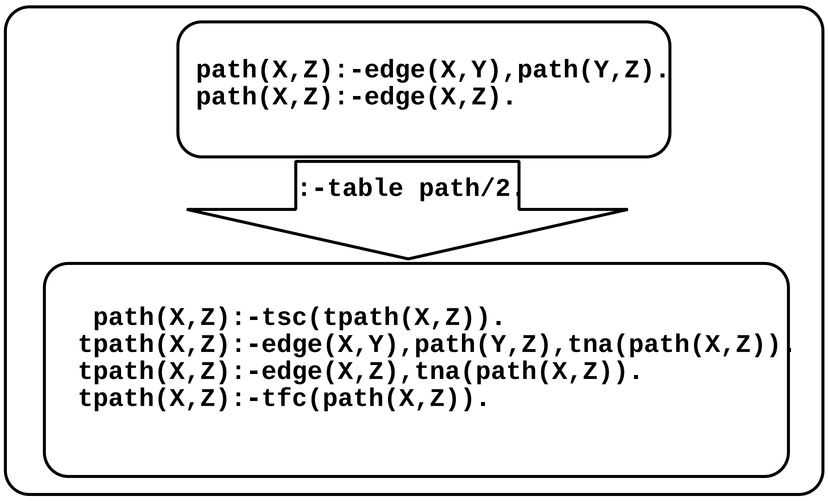
Concerning the compilation of tabled logic programs, when a tabled program is loaded in a Prolog system supporting tabling, the parsing phase will search for *table p/n* declarations. These declarations indicate that calls to predicate *p/n* are to be executed using tabled evaluation. Thus, these predicates are compiled with specific tabling instructions that will allow the tabling component of the system to have extra control over the program's flow of execution. The most important tabling instructions are:

- **Tabled Subgoal Call (tsc)**. Checks if a call is the first call for a subgoal. If so, it allocates a generator node and adds a new entry to the table space. If the subgoal is already in the table space, this means that it is not the first call, so this instruction allocates a consumer node and resolves the subgoal by consuming the available answers.

- **Tabled New Answer (tna)**. Checks if an answer found for a particular subgoal is new or repeated. If the answer is new, it is inserted in the table space and the evaluation proceeds accordingly with the scheduling strategy (this will be discussed in more detail next). Otherwise, if the answer is repeated, the operation fails.

- **Tabled Fix-point Check (tfc)**. Determines whether a fix-point was reached. When this is the case, the subgoal is marked as completed, otherwise it will be scheduled for another re-evaluation. A fix-point is reached when no unexploited answers are available for consumers and generators can not produce any new answers.

The **Tabled Subgoal Call** instructions are an extension of the original WAM choice point instructions, while the **Tabled New Answer** and **Tabled Fix-point Check** instructions were created exclusively for tabling support. Using this terminology, Fig. 2.7 shows the generic transformation of the original $path_1$ program into a program using tabled evaluation.



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

```
:-table path/2.
```

```
 path(X,Z):-tsc(tpath(X,Z)).
tpath(X,Z):-edge(X,Y),path(Y,Z),tna(path(X,Z)).
tpath(X,Z):-edge(X,Z),tna(path(X,Z)).
tpath(X,Z):-tfc(path(X,Z)).
```

Figure 2.7: Generic tabled transformation for the $path_1$ program

After transformation, the `path/2` predicate remains only on one clause, this clause will work as an entry point to the new auxiliary predicate (`tpath/2`) representing the transformed predicate. This new predicate holds three clauses, the first two are the extension of the original `path/2` clauses with the `tna` instruction (this will allow the detection of all answers found on each clause) and the third clause is used to execute the `tfc` instruction.

2.4.2 Table Space

The table space is a key component of a tabling engine. The overall performance of a tabling system can be directly affected, if the basic operations that manipulate the table space are not implemented efficiently. Typically, the table space can be accessed to look up for tabled subgoals (and tabled answers) and to consume answers present on each tabled subgoal. Currently there are two major implementations: the B-Prolog system uses *hash tables* [39] and the YapTab and XSB Prolog systems, use tries [26] based on the proposal made by I. V. Ramakrishnan et al. [21, 22]. The *hash tables* are expected to be slower than tries for complex terms, since tries provide a complete discrimination of terms, permitting the lookup and possibly insertion to be performed in a single pass through a term [39].

Lets now analyze in more detail, how the tabling engine interacts with the table space. When a tabled call is made, the first operation is to ground the call. This grounding of the call makes it possible to distinguish between first calls and repeated calls to the same predicate. Figure 2.8 shows some grounding examples for a `path/3` predicate. The non-variable terms present on the predicate remain unchanged, but the variables are abstracted and numbered by order of appearance.

Original Tabled Call		**Grounded Tabled Call**
`path(X,Y,Z)`	→	`path(VAR0,VAR1,VAR2)`
`path(a,X,Y)`	→	`path(a,VAR0,VAR1)`
`path(X,X,X)`	→	`path(VAR0,VAR0,VAR0)`
`path(f(X),Y,X)`	→	`path(f(VAR0),VAR1,VAR0)`
`path(a,b,c)`	→	`path(a,b,c)`

Figure 2.8: Grounding examples for a path/3 predicate

Then, the next step is to integrate the grounded call on the table space. The integration depends on whether the call is made via the `tsc` instruction or via the `tna` instruction. For the `tsc` instruction, the tabling engine performs a search over the calls already in table space in order to check if the call is already there. If it is a first call then a new entry is created. Otherwise, it is a repeated call, so the call is scheduled for answer consumption. For the `tna` instruction, the tabling engine searches the answers in the table space for the corresponding tabled call and if it is a new answer, it is added to table space. Using YapTab's organization based on tries, Fig. 2.9 shows the table space organization for a `path/3` predicate.

The table space is organized in two level of tries. The top level (subgoal trie structure) is used to store the subgoal calls. The lower level (answer trie structures) is used to store the answers for each subgoal call.

2.4.3 Suspended Tabling vs Linear Tabling

The implementation of tabling engines on Prolog systems is actually based in two major paradigms: *suspension-based* tabling and *linear-based* tabling.

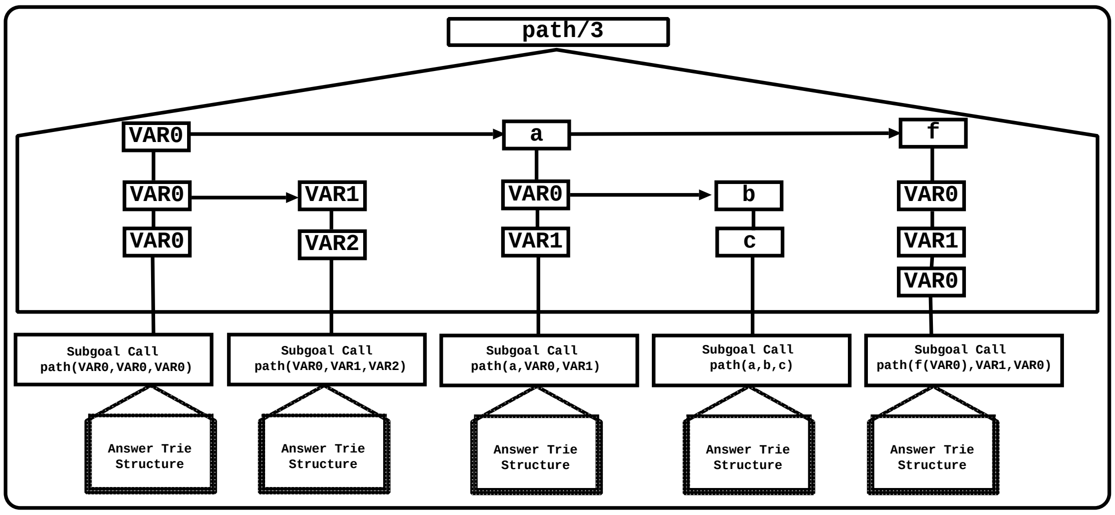


Figure 2.9: Using tries to represent the table space organization of a path/3 predicate

Suspension-based tabling mechanisms keep the execution stacks, of the sub-computations corresponding to consumer nodes, in order to resume them as new answers are found for the tabled subgoals involved on those sub-computations. Since this mechanism avoids the evaluation steps required to put the computation on the same state where those sub-computations were suspended (as it can directly restore the suspended stacks), it has the advantage of reducing the execution time of a program. There are however two major drawbacks for this mechanism. The first, is that it is considered to be hard to implement. The second is on the memory side, as it requires extra usage of memory on stacks. In fact, as memory resources are finite, it is possible to have programs with intensive usage of memory, that cannot be computed because of the extra burden caused by the additional resources needed to keep all the sub-computations on the stacks. Since the first implementation of a suspension based mechanism [29], different approaches were found to reduce memory overheads. The Mercury implementation [32] and two alternative XSB-based models, the CAT [9] and the CHAT [10] models, copy the execution stacks to a separate storage place. Two more recent approaches, implemented in Yap [27] and Ciao Prolog [6], feature a higher-level implementation of suspension-based tabling. They apply source level transformations to a tabled program and then use external tabling primitives to provide direct control over the search strategy.

On the other hand, the linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The basic idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. Two different optimization proposals are the SLDT strategy of Zhou *et al.* [39], as originally implemented in B-Prolog, and the DRA technique of Guo and Gupta [12], as originally implemented in ALS-Prolog. The key idea of the SLDT strategy is to let repeated calls execute from the backtracking point of the former call. The repeated call is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a fix-point is reached. Current versions of B-Prolog implement an optimized variant of this strategy which tries to avoid re-evaluation of looping subgoals [38]. The DRA technique is based on dynamic reordering of alternatives with repeated calls. This technique tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls (*looping alternatives*). It then uses the looping alternatives to repeatedly recompute them until reaching a fix-point. We will discuss these optimizations in more detail in the following chapters.

2.4.4 Batched Scheduling vs Local Scheduling

The decision about the evaluation flow is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *batched scheduling* and *local scheduling* [11].

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. It favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the execution continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC (*Strongly Connected Component*) [29] and delaying the completion point to an older generator node.

On the other hand, the local scheduling strategy allows a cluster of subgoals to return answers only after the fix-point has been reached [11]. In other words, only one SCC is evaluated at each time. Whenever a new answer is found, it is added to the table space and the computation fails to the top most choice point, which is the one that is being evaluated. Tabled subgoals inside an SCC only propagate their answers to outside the SCC, after their completion, which occurs when SCC's fix-point is found. Local scheduling causes a sooner completion of subgoals, which creates less complex dependencies between them. We will discuss both strategies in more detail next.

2.4.5 An Example of a Tabled Evaluation

In order to completely understand how the tabling mechanism changes the evaluation flow of a program in order to overcame the problem of recursive calls to the same subgoal, we next show on Fig. 2.10, the tabled evaluation of the $path_1$ program with the $edge_1$ transition function, using linear tabling with local scheduling. The figure is divided into three different areas:

- **The tabled program**. It shows the transformed program of Fig. 2.7 in conjunction with a graph defined by two *edge*/2 facts. The *c1, c2, c3, c4, c5* and *c6* labels represent the six program clauses.
- **The table space**. It shows all the subgoal calls and answers found during the evaluation. The additional field *state* is used to distinguish between the three possible situations a subgoal can be during evaluation: when the subgoal has to explore the program clauses (*ready*), when it has to consume answers from table space (*evaluating*) and when it is completely evaluated (*complete*).
- **The evaluation tree**. It details all the computational steps of the program's evaluation. Black oval boxes represent generator choice points and white oval boxes represent consumer choice points. The numbers on the left side of two dots, indicate the current computational step.

Lets analyze the most important computational steps of the example. The evaluation starts with the query call `path(1,Z)`. Since this is the first appearance of subgoal `path(1,Z)`, the `tsc` instruction allocates a new generator choice point, adds an entry for this subgoal to the table space and sets the subgoal's state to `evaluating` (step 1). The evaluation proceeds as a standard Prolog system, which means using a depth-first left-most strategy. So, it proceeds with the exploration of the first matching clause, which is `c2`. Step 2 explores the non-tabled predicate `edge/2` and the evaluation reaches to the subgoal call `path(2,Z)`. As this is the first appearance of the subgoal `path(2,Z)`, a new generator choice point is allocated and a new entry is added to the table space (step 3). On step 5, the evaluation reaches to a repeated call of subgoal `path(1,Z)`. In this situation, the `tsc` instruction marks `path(2,Z)` as depending on `path(1,Z)` and allocates a new consumer node. But as `path(1,Z)` does not have any

```
 path(X,Z):-tsc(tpath(X,Z)).                          (c1)
tpath(X,Z):-edge(X,Y),path(Y,Z),tna(path(X,Z)).  (c2)
tpath(X,Z):-edge(X,Z),tna(path(X,Z)).            (c3)
tpath(X,Z):-tfc(path(X,Z)).                      (c4)
edge(1,2).                                       (c5)
edge(2,1).                                       (c6)
```

Call	Solutions	State
1: path(1,Z)	11: Z=1 14: Z=2	1: evaluating 56: complete
3: path(2,Z)	8: Z=1 23: Z=2	3: evaluating 16: ready 18: evaluating 36: ready 56: complete

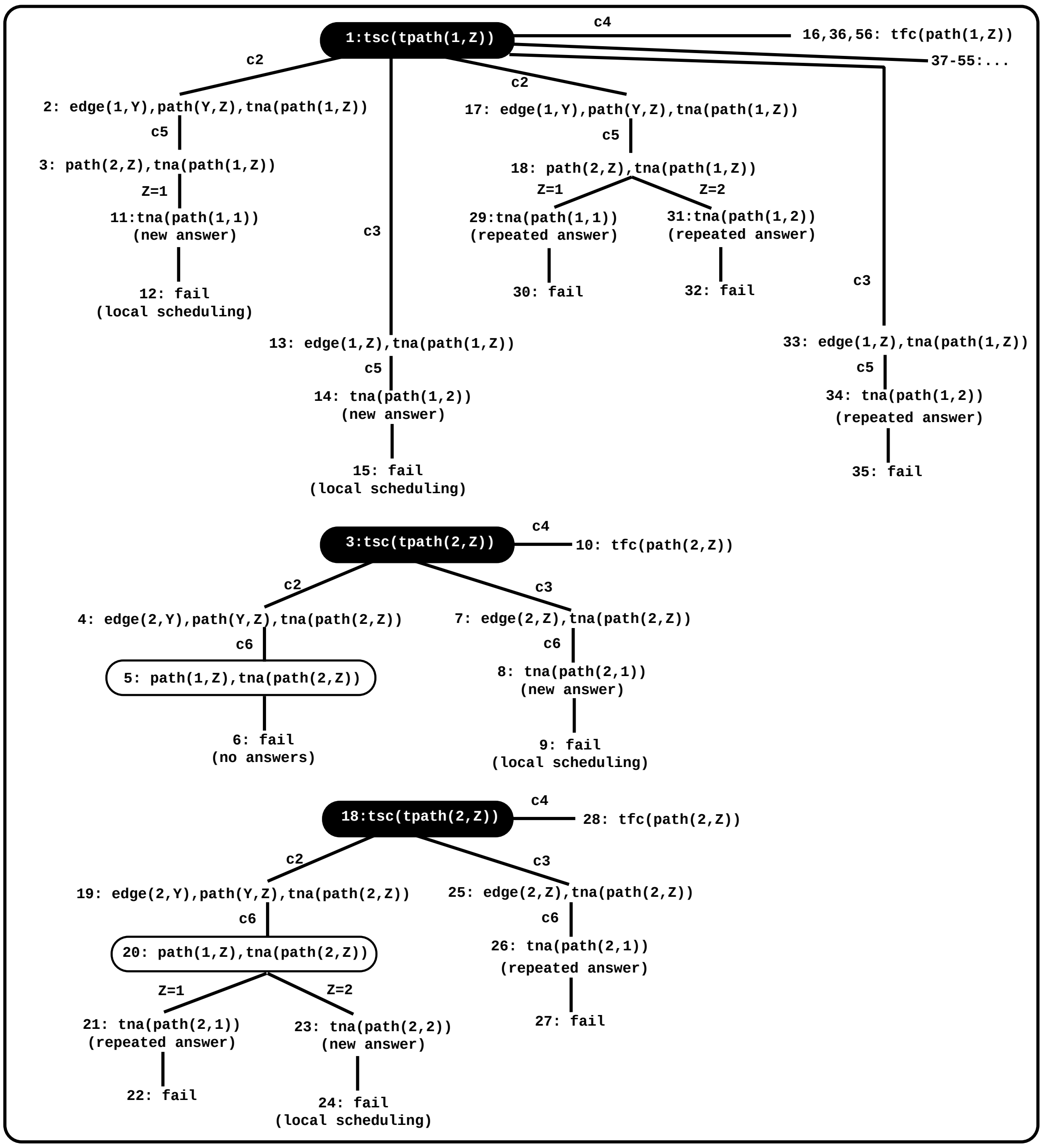


Figure 2.10: Evaluation of $path_1$ program using linear tabling with local scheduling

answer on the table space, the computation fails (step 6). As the top most choice point is a consumer node, the backtracking mechanism pops off the choice point and marks the second matching clause of the subgoal `path(2,Z)` for exploration. The clause `c3` is explored and the `tna` instruction adds the first answer (`Z=1`) to the table entry of the subgoal `path(2,Z)` (step 8). Since the evaluation is using local scheduling, the `tna` instruction fails (step 9) and the computation jumps to the fix-point check instruction (step 10). The `tfc` instruction checks if `path(2,Z)` depends on any other subgoal, and as this is the case, it converts the generator choice point into a consumer choice point and starts consuming the available answers on the table space for `path(2,Z)`. The answer `Z=1` is thus propagated to subgoal `path(1,Z)` and the `tna` instruction inserts it in the table entry for `path(1,Z)` (step 11) and fails again for the same reason (step 12). The second matching clause for `path(1,Z)` is explored on step 13 and the answer `Z=2` is found. The `tna` instruction fails again (step 15), so evaluation checks for a fix-point (step 16). Since subgoal `path(2,Z)` depends on `path(1,Z)`, the subgoal `path(1,Z)` is the leader of the SCC. This means that subgoal `path(1,Z)` is responsible for completing the SCC or scheduling it for a new re-evaluation. As new answers were found during the current round, the `tfc` instruction marks the SCC for a new re-evaluation and sets the state of subgoal `path(2,Z)` as `ready`. On step 18 the state of `path(2,Z)` moves again to `evaluating` and on step 23, the answer `Z=2` is found and added to table space of subgoal `path(2,Z)`. On step 36, the SCC is re-scheduled for another re-evaluation. This last round does not find any new answer, so the subgoal `path(1,Z)` completes the SCC by marking both subgoals as `complete`. The evaluation of $path_1$ program is thus completed at step 56. All the branches of the search space were completely explored and consequently all the answers were found.

2.5 Chapter Summary

This chapter introduced several important concepts about Logic Programming and the implementation of Prolog systems. It discussed some well-known and important limitations of Prolog systems in order to motivate for the appearance of tabling mechanisms. Focus was then given to tabled evaluations, using different scheduling policies and paradigms which will be described on the next chapters in more detail.

Chapter 3

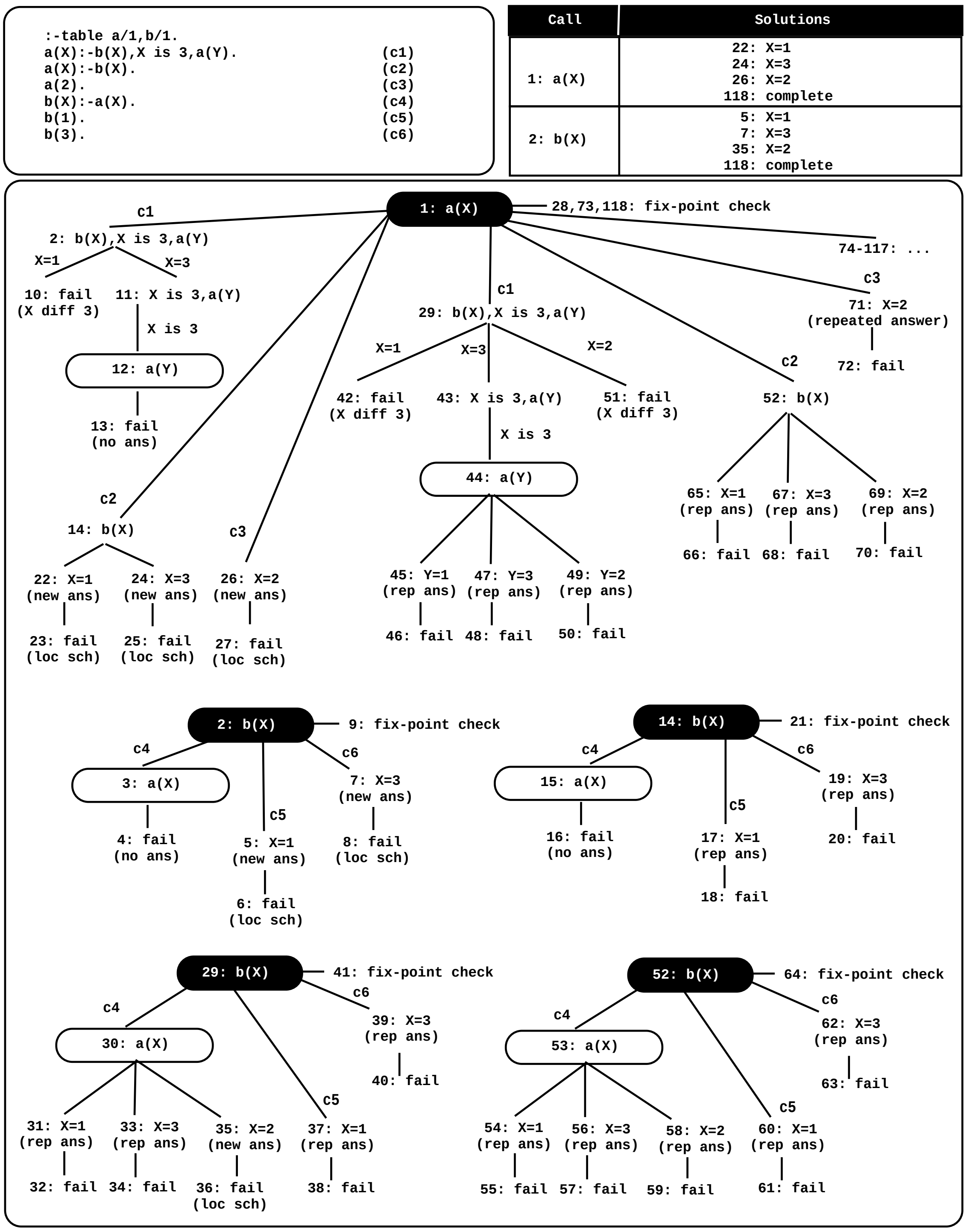
Linear Tabling Strategies

This chapter describes the key ideas behind standard linear tabled evaluation and its optimizations. The first section introduces the standard approach of linear tabling and the next sections present four different optimizations, that can be used alone or combined with each other, to reduce the search space of the standard approach. The first optimization tries to reduce the total number of choice points during the evaluation, the second and third try to reduce the total number of branches to be explored (alternatives and answers respectively) and the last optimization, concentrates on dynamically reordering the evaluation. As the focus of the chapter is to describe the key ideas behind these strategies, all the low-level details will be left aside for the next chapter.

3.1 Standard Linear Tabling

The standard linear tabling mechanism uses a naive approach to evaluate tabled logic programs. Every time a new answer is found during the last round of an evaluation over the current SCC, the complete search space of the SCC is schedule for a new round of re-evaluation. Figure 3.1 presents how standard linear tabling works through an example.

The example corresponds to the evaluation of a logic program, which we called FTS (Find the Three Solutions), that we will use during this chapter to introduce our optimizations. The objective of the FTS program is to get all the solutions for the query call $a(X)$. The solution set of the program is $\{X = 1,\ X = 2,\ X = 3\}$. The program was specifically designed to create a small SCC with the necessary conditions to apply the optimizations which will be presented on the next sections. First, lets analyze its evaluation, using standard linear tabling with local scheduling strategy.



```
:-table a/1,b/1.
a(X):-b(X),X is 3,a(Y).          (c1)
a(X):-b(X).                      (c2)
a(2).                            (c3)
b(X):-a(X).                      (c4)
b(1).                            (c5)
b(3).                            (c6)
```

Call	Solutions
1: a(X)	22: X=1 24: X=3 26: X=2 118: complete
2: b(X)	5: X=1 7: X=3 35: X=2 118: complete

Figure 3.1: A standard linear tabled evaluation of the FTS program

We start the evaluation of the FTS program, by inserting a new entry in the table space representing the generator call $a(X)$ (step 1). Generator calls are depicted by black oval boxes. Then, the subgoal $a(X)$ is resolved against its first matching clause ($c1$), calling the subgoal $b(X)$ in the continuation. As this is a first call to $b(X)$, a new entry is inserted in the table space representing $b(X)$ and we proceed as shown in the left below tree (step 2). The subgoal $b(X)$ is also resolved against its first matching clause ($c4$), calling again $a(X)$ in the continuation (step 3). Since $a(X)$ is a repeated call, we allocate a consumer node and try to consume answers from the table space. Consumer calls are depicted by white oval boxes. But at this stage no answers are available, so execution fails (step 4).

We then try the second matching clause ($c5$) for the subgoal $b(X)$, the answer $X = 1$ is found and added to table space of subgoal $b(X)$ (step 5). Next, as we are following a *local scheduling* strategy, the execution *fails* [11]. With local scheduling, new answers are only returned to the calling environment when all program clauses were explored. The execution thus fails back to node 2 and we start exploring the third matching clause ($c6$) for subgoal $b(X)$. The answer $X = 3$ is found and added to table space (step 7), and since we are following a *local scheduling* strategy, we fail again (step 8). As there aren't any more matching clauses for the subgoal $b(X)$, we check for a *fix-point* (step 9), but the subgoal $b(X)$ is not a leader call because it has a dependency (consumer node 3) to the older subgoal $a(X)$. Remember that an SCC reaches a fix-point when no new answers are found for the leader subgoal, during the last round of evaluation.

Next, we propagate the answers of $b(X)$ to the context of the previous call, so the answers $X = 1$ and $X = 3$ are propagated to node 2. The first answer, $X = 1$, does not match the test `X is 3`, so the computation fails (step 10). But for the second answer, the test `X is 3` succeeds, so the computation advances and calls the subgoal $a(Y)$, which is a variant call of subgoal $a(X)$ (step 12). We allocate a new consumer node in order to consume the answers on the table space of $a(X)$, but at this stage no answers are still available, so the execution fails (step 13) to node 1. We then try to explore the second matching clause ($c2$) for the subgoal $a(X)$ which calls again the subgoal $b(X)$ (step 14). A new generator choice point is allocated for subgoal $b(X)$ and we schedule again the three matching clauses $c4$, $c5$ and $c6$, but no new answers are found (steps 15 to 21). We start then, propagating the answers of subgoal $b(X)$ to the previous call and answers $X = 1$ and $X = 3$ are found and added to the table space of subgoal $a(X)$ (steps 22 to 25). Since we are following local scheduling, we fail the computation to node 1 and start exploring the third matching clause ($c3$) of subgoal

$a(X)$. This leads to the new answer $X = 2$, which is also added to the table space of $a(X)$, and the computation fails. Since $c3$ is the last matching clause for subgoal $a(X)$, we check if a fix-point for the SCC was reached (step 28), but as new answers were found for subgoals $a(X)$ and $b(X)$, we schedule the SCC for a new re-evaluation.

On the new round, we repeat the same sequence as in steps 2 to 27 (now steps 29 to 72). The difference is that now subgoal $a(X)$ has three answers in its table space. However, only a new answer $X = 2$ is found for subgoal $b(X)$ and added to its table space at step 35. All the remaining answers found during the current round are repeated. On step 73 we check again for a fix-point, but due to the answer found on step 35, it was not reached yet, so we schedule the SCC for a new re-evaluation. This round of evaluation does not find any new answer (steps 74 to 117) for both subgoals, so we have finally achieved a fix-point. We complete the SCC, by marking the subgoals $a(X)$ and $b(X)$ as *complete* and the program's evaluation is finished (step 118). All the answers inside the solution's set were successfully found.

Figure 3.2 shows snapshots of the local stack during the evaluation of the example in Fig. 3.1. For example, Fig. 3.2(a) shows the local stack configuration at the end of the first three steps. The first calls to subgoals $a(X)$ and $b(X)$ have a generator choice point allocated on the local stack and the second call to subgoal $b(X)$ has a consumer choice point.

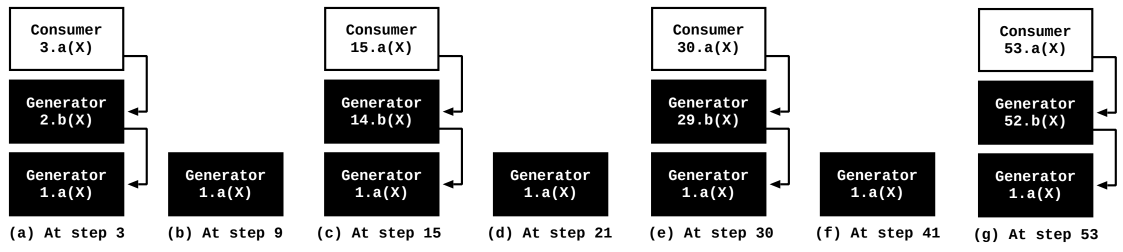


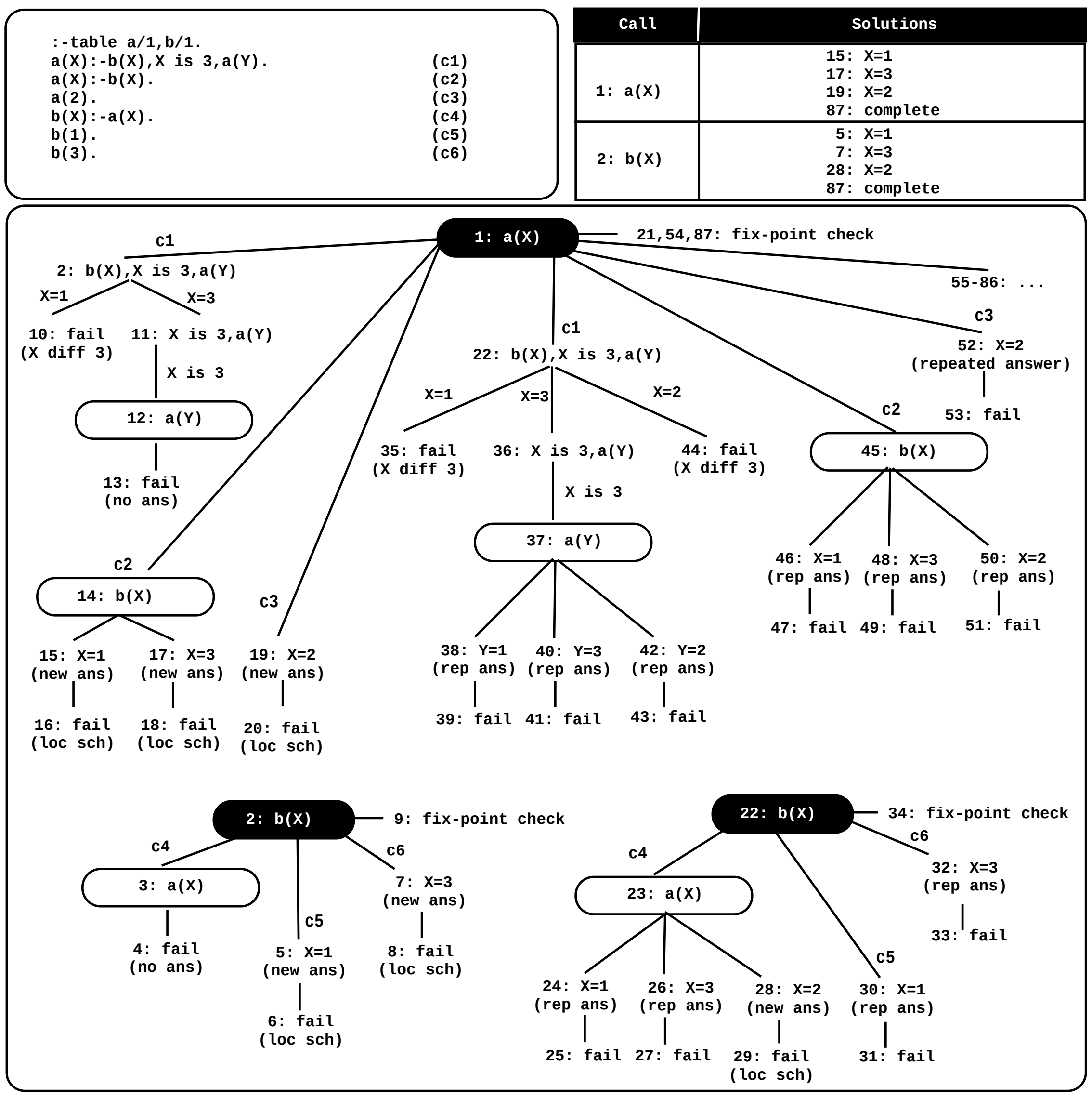
Figure 3.2: Local stack configuration for the evaluation of Fig. 3.1

If we analyze the general behavior of the local stack during the evaluation of the FTS program, we can observe that it presents a sinusoidal aspect. The maximum is achieved when we have two generator choice points and one consumer choice point allocated at the same time (Fig. 3.2(a), (c), (e) and (g)), and the minimum is achieved when we only have one generator choice point (Fig. 3.2(b), (d) and (f)). The next section presents an optimization to the standard linear tabling that tries to reduce the number of times that this maximum is achieved.

3.2 Eliminating Repeated Generator Calls

We next describe the first of four optimizations implemented on top of the standard linear engine. The main principle of this first optimization is to reduce the search space exploration by executing only once the same sub-computation inside each SCC round of evaluation. Analyzing the sinusoidal aspect of Fig. 3.2, if we center our attention to what happens to subgoal $b(X)$, we can observe that first calls to it inside the SCC are detailed on Fig. 3.2(a) and (e), and that repeated calls are detailed on Fig. 3.2(c) and (g). Now matching the computations on Fig. 3.1 which correspond to first calls against the repeated calls to subgoal $b(X)$, it is possible to observe that repeated calls always lead to the same computation of the first calls. This means that, executing program clauses which match with repeated calls, most of the times, won't lead to further developments on the table space for the subgoals at hand. Thus, we have generalized this observation and created the optimization *Eliminating Repeated Generator Calls (ERGC)*. The objective of this optimization is to avoid redundant computations, by scheduling the re-evaluation of non-leader tabled calls, in such a way that the number of allocated choice points is reduced to a minimum, i.e., in each evaluation round only the first calls to tabled subgoals allocate generator choice points to execute alternatives. All the remaining calls allocate consumer choice points [3].

Figure 3.3 shows a new evaluation of the FTS program, using the ERGC optimization. For this evaluation, for each round of evaluation over the SCC, we will use generator nodes only on the first calls of each subgoal. So we begin the evaluation, as expected, by allocating a generator node for the subgoal $a(X)$ (step 1). Next, we start evaluating the first matching clause of the subgoal $a(X)$, which leads to the first call to the subgoal $b(X)$ (step 2). We then proceed as for the standard evaluation until reaching the second matching clause of subgoal $a(X)$ (step 14). At this step, we have a repeated call to subgoal $b(X)$, so we will allocate a consumer choice point instead of a generator choice point. Since we do not explore the program clauses at this step, we start consuming the available answers on the table space of $b(X)$. This leads to finding answers $X = 1$ and $X = 3$ for subgoal $a(X)$, sooner than with the standard evaluation (the current evaluation finds these answers on steps 15 and 17, while the previous evaluation found them on steps 22 and 24, respectively). We proceed then, as for the standard evaluation until step 45. At this step, we have a new repeated call to subgoal $b(X)$ (the first call to $b(X)$ on the current round was made at step 22), so we allocate again another consumer choice point and start consuming the answers on the table space of the subgoal $b(X)$ instead of executing the program clauses. Finally, we conclude the evaluation of the FTS program on step 87, which is earlier than the previous standard evaluation which was only concluded on step 118.



```
:-table a/1,b/1.
a(X):-b(X),X is 3,a(Y).        (c1)
a(X):-b(X).                    (c2)
a(2).                          (c3)
b(X):-a(X).                    (c4)
b(1).                          (c5)
b(3).                          (c6)
```

Call	Solutions
1: a(X)	15: X=1 17: X=3 19: X=2 87: complete
2: b(X)	5: X=1 7: X=3 28: X=2 87: complete

Figure 3.3: Evaluation of the FTS program using the ERGC optimization

The advantages of this optimization are obvious. It has reduced the total number of evaluation steps in 29 steps and as Fig. 3.4 shows, it has reduced also the usage of the local stack.

Figure 3.4 presents again the sinusoidal aspect observed in Fig. 3.2. The maximum number of choice points still corresponds to two generator choice points and one consumer choice point, but was achieved less times. Moreover, Fig. 3.4(c) and (g), which correspond to the repeated calls to subgoal $b(X)$, show less expansion of the local

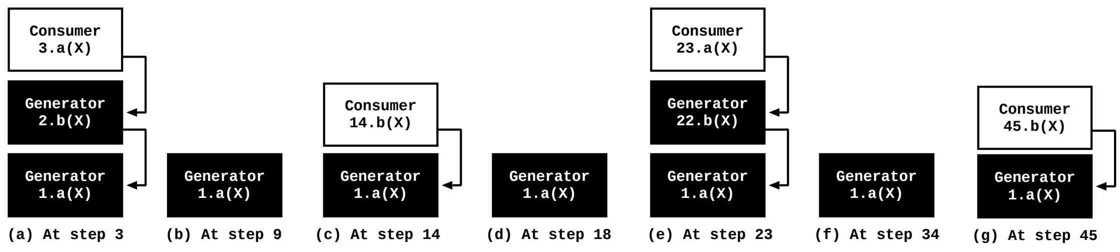


Figure 3.4: Local stack configuration for the evaluation of Fig. 3.3

stack. Consequently, during each round of evaluation over the SCC, the maximum number of choice points is achieved only once, instead of two times as with the standard evaluation.

3.3 Dynamic Reordering of Alternatives

The DRA linear tabling mechanism as originally proposed by Guo and Gupta [12] is based on the *dynamic reordering of alternatives with repeated calls* for incorporating tabling into an existing logic programming system. The DRA technique not only memorizes the answers for the tabled subgoal calls, but also the alternatives leading to repeated calls, the *looping alternatives*. It then uses the looping alternatives to repeatedly recompute them until a fix-point is reached. During evaluation, a tabled call can be in one of three possible states: *normal*, *looping* or *complete*. Figure 3.5 shows the state transition graph for DRA evaluation.

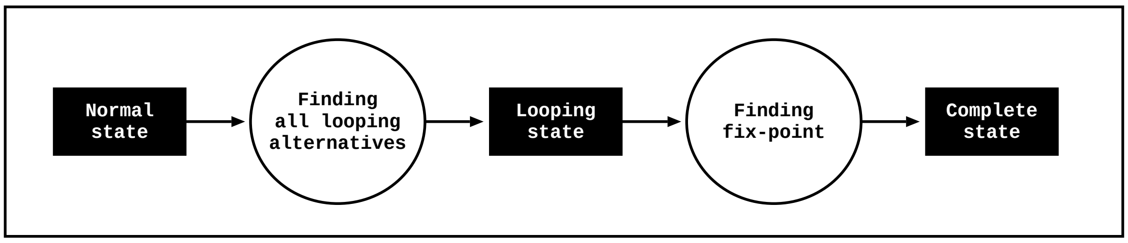


Figure 3.5: State transition graph for DRA evaluation

Consider a tabled subgoal call C. Initially, C enters in normal state where it is allowed to explore the matching clauses as in a standard evaluation. In this state, while exploring the matching clauses, the model checks for looping alternatives. If a repeated (or variant) call is found, then the current clause for the first call to C will be memorized as a looping alternative. Essentially, the alternative corresponding to this call will be reordered and placed at the end of the alternative list for the call. As in a tabled evaluation, repeated calls are not re-evaluated against the program clauses because they can potentially lead to infinite loops, the repeated call to C is thus resolved by consuming the answers already available for the call in the table space. Next, after exploring all the matching clauses, C goes into the looping state. From this point on, it keeps trying the looping alternatives repeatedly until reaching a fix-point. If no new answers are found during one cycle of trying the looping alternatives, then the evaluation has reached a fix-point and C is completely evaluated. However, if C is inside an SCC, then completion is only performed at the leader call, as discussed previously.

Figure 3.6 shows the evaluation sequence of the FTS program, using the DRA optimization combined with the ERGC optimization presented on the previous section. The figure has a new field in the table space called *Loop Alt*, which is used to store the looping alternatives of each subgoal.

As DRA uses the first round of evaluation over the SCC to detect the looping alternatives, we evaluate the first round in a similar manner to the evaluation of Fig. 3.3. The difference is that at step 3, when we detect the first looping call, we add the current alternative in evaluation for each subgoal to the respective table space. This means that at step 3, we store the clause $c1$ on the table space of subgoal $a(X)$ and the clause $c4$ on the table space of subgoal $b(X)$. We then proceed with the evaluation up to step 14, where we detect another looping call, this time to subgoal $b(X)$ and the clause $c2$ is stored as a looping alternative for the subgoal $a(X)$. At step 21, we finish the first round of evaluation over the SCC and schedule the SCC for a new round.

In this new round, we will only evaluate the looping alternatives, this means that the remaining alternatives won't be evaluated. So, we start the second round by evaluating the clause $c1$ for $a(X)$ which leads to the first call of subgoal $b(X)$ (step 22). The subgoal $b(X)$ only has one looping alternative (clause $c4$), so we evaluate it and in consequence we achieve the solution $X = 2$, which is added to the table space of subgoal $b(X)$. As we are following local scheduling, the evaluation then fails to node 22 and at this stage, we avoid executing the program clauses $c5$ and $c6$, since they were not marked as looping, so at step 30 we check for a fix-point. Later, at step 40, when the evaluation fails to node 1, we execute the second looping alternative, clause $c2$, and at step 47, when the evaluation fails again to node 1, we avoid executing the program clause $c3$, since it is not marked as looping and we check for a new fix-point (step 48). Since the fix-point was not reached, we schedule the SCC for a new re-evaluation round. We conclude the evaluation of the FTS program on step 75, which is earlier than the previous evaluation of Fig. 3.3 which was concluded on 87 steps.

```
:-table a/1,b/1.
a(X):-b(X),X is 3,a(Y).                    (c1)
a(X):-b(X).                                (c2)
a(2).                                      (c3)
b(X):-a(X).                                (c4)
b(1).                                      (c5)
b(3).                                      (c6)
```

Call	Solutions	Loop Alt
1: a(X)	15: X=1 17: X=3 19: X=2 75: complete	3: c1 14: c2
2: b(X)	5: X=1 7: X=3 28: X=2 75: complete	3: c4

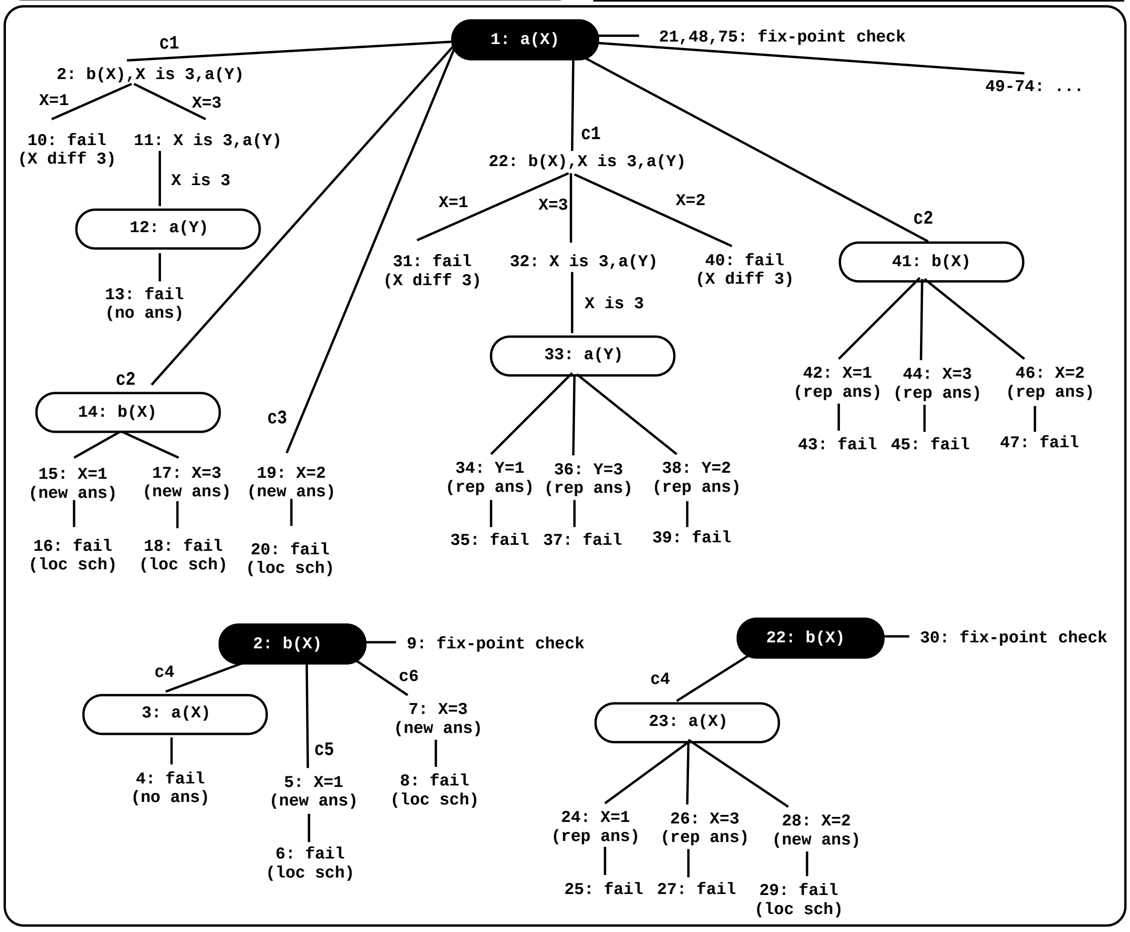


Figure 3.6: Evaluation of the FTS program using the combination of DRA with ERGC

In summary, for the FTS program with the DRA and ERGC optimizations, the search space of the first round is similar to the evaluations described previously, but the search space of the next two rounds was reduced because the clauses $c3$, $c5$ and $c6$ were avoided. Due to this fact, the total number of evaluation steps was reduced in 12 steps.

3.4 Dynamic Reordering of Solutions

The third optimization, is called *Dynamic Reordering of Solutions (DRS)*, and can be seen as a variant of the DRA optimization, because instead of reordering the consumption of alternatives, it reorders the consumption of solutions.

The main idea of the DRS strategy is to store the solutions leading to consumer calls, the *looping solutions*. When a non-leader generator call C consumes solutions to propagate them to the context of the previous call, if a consumer call is found, the current solution for C is memorized as a looping solution. Later, if C is scheduled for re-evaluation, instead of trying the full set of solutions, it only tries the looping solutions plus the new solutions found during the current round. In each round, the new solutions leading to consumer calls are added to the previous set of looping solutions.

Figure 3.7 shows another evaluation for the FTS program, this time using the combination of the DRS, DRA and ERGC optimizations. In order to support the storage of looping solutions for each subgoal, the table space was extended with a new field called *Loop Sol*.

On the first round of evaluation over the SCC, we start the evaluation as usual by allocating a generator choice point for the query call $a(X)$. At steps 5 and 7, the solutions $X = 1$ and $X = 3$ are found and added to the table space of subgoal $b(X)$ and at step 10, we start propagating them to the previous call. For the solution $X = 1$, the evaluation fails because X is different from 3. For the solution, $X = 3$, the test `X is 3` succeeds and the evaluation advances to the next goal, which originates a consumer call to the subgoal $a(Y)$ (step 12). At this step we thus mark the solution $X = 3$ as a looping solution, by adding it to the table space of the subgoal $b(X)$.

The first round of evaluation is then completed at step 21 and at step 22 we start a second round. At step 28, the solution $X = 2$ is found for the subgoal $b(X)$ and added to its table space and, at step 31, we start propagating the solutions for subgoal $b(X)$ to the context of the previous call. We begin with the looping solutions instead of the new solutions, because we want to avoid consuming more than once the same solution on the same stage of the computation.[1] We skip the solution $X = 1$, since it was not

[1]Suppose that during the current round, we found a new solution NS. Now consider that we started by consuming the solution NS and that this solution leads to a consumer call, then we would mark it as a looping solution. As after consuming all the new solutions, we consume the looping solutions, we would consume the solution NS twice.

```
:-table a/1,b/1.
a(X):-b(X),X is 3,a(Y).                 (c1)
a(X):-b(X).                             (c2)
a(2).                                   (c3)
b(X):-a(X).                             (c4)
b(1).                                   (c5)
b(3).                                   (c6)
```

Call	Solutions	Loop Alt	Loop Sol
1: a(X)	15: X=1 17: X=3 19: X=2 72: complete	3: c1 14: c2	
2: b(X)	5: X=1 7: X=3 28: X=2 72: complete	3: c4	12: X=3

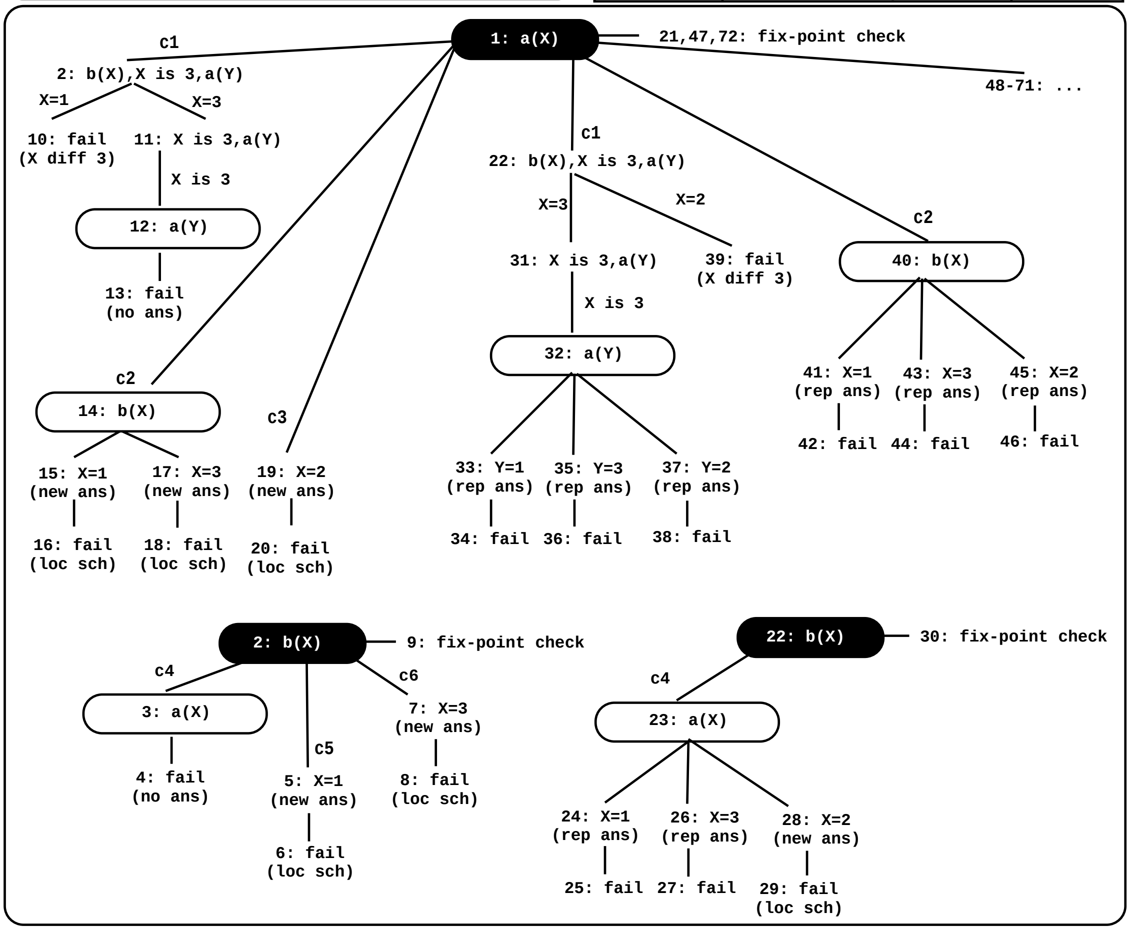


Figure 3.7: Evaluation of the FTS program using the combination of DRS, DRA and ERGC

marked as a looping solution, and we start by consuming the solution $X = 3$, which won't lead to any further developments on the table space (step 31). The same occurs when the solution $X = 2$ is consumed (step 39). The current round of evaluation over the SCC is concluded on step 47 and, at the step 48, we start the third and last round (steps 48 to 71). On this last round, the generator call of the subgoal $b(X)$, propagates only the looping answer $X = 3$ to the previous call, since no new answers will be found. This means that the evaluation of FTS program is concluded in 72 steps, which is earlier that the previous evaluation that was concluded on 75 steps.

In summary, for the FTS program with the DRS, DRA and ERGC optimizations, the search space of the first round is similar to the evaluation discussed previously, but the search space of the next two rounds was slightly reduced because the answers $X = 1$ and $X = 2$ on the generator calls of the subgoal $b(X)$ were not consumed. Due to this fact, the total number of evaluation's steps was reduced in 3 steps.

3.5 Dynamic Reordering of Execution

The last optimization called *Dynamic Reordering of Execution (DRE)*, is based on the original SLDT strategy, as proposed by Zhou et al. [39]. The key idea of the DRE strategy is to let repeated calls to tabled subgoals execute from the *backtracking clause of the former call*. A first call to a tabled subgoal is called a *pioneer* and repeated calls are called *followers* of the pioneer. When backtracking to a pioneer or a follower, we use the same strategy, first we explore the remaining clauses and only then we try to consume answers. The fix-point check operation is still performed only by pioneer calls.

Figure 3.8 shows the last evaluation of the FTS program. For this evaluation, we will combine the DRE optimization with the DRS, DRA and ERGC optimizations. No extra fields are added to the.

As for the previous examples, we start the evaluation with the first matching clause ($c1$) for the first (pioneer) call to subgoal $a(X)$ (step 1). The subgoal $b(X)$ is called in the continuation (step 2) and the subgoal $a(X)$ is then called repeatedly (step 3). But now, as we are using the DRE optimization, the subgoal $a(X)$ is considered a follower and thus it *steals* the backtracking clause of the former call at node 1, i.e., the second matching clause ($c2$) for the subgoal $a(X)$. We thus proceed the evaluation, as if it was a generator call (this means that new answers can be generated for the subgoal $a(X)$). We thus evaluate the clause $c2$ and the subgoal $b(X)$ is called again (step 4). Since subgoal $b(X)$ has its pioneer call on node 2, we act as a follower and start the evaluation of the next clause, clause $c5$ (step 5). The answers $X = 1$ and $X = 3$ are found and added to the table space of subgoal $b(X)$ (steps 5 to 8).

As we are following a local scheduling strategy, we fail and backtrack to the follower node 4. As all matching clauses of $b(X)$ were already evaluated, we proceed the evaluation by propagating the answers of the subgoal $b(X)$ to the previous call. The answers $X = 1$ and $X = 2$ are then added to the table space of subgoal $a(X)$ (steps

```
:-table a/1,b/1.
a(X):-b(X),X is 3,a(Y).             (c1)
a(X):-b(X).                         (c2)
a(2).                               (c3)
b(X):-a(X).                         (c4)
b(1).                               (c5)
b(3).                               (c6)
```

Call	Solutions	Loop Alt	Loop Sol
1: a(X)	9: X=1 11: X=3 13: X=2 57: complete	3: c1 4: c2	
2: b(X)	5: X=1 7: X=3 19: X=2 57: complete	3: c4	23: X=3

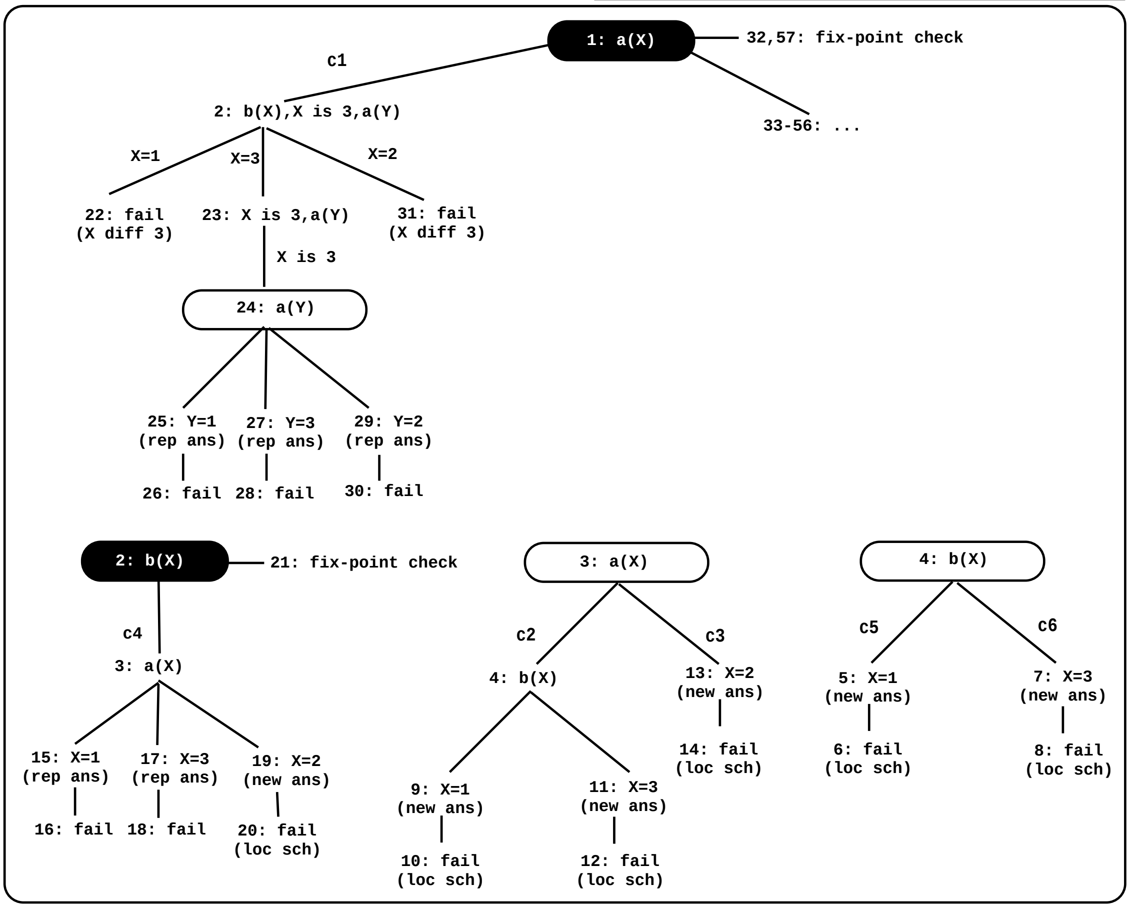


Figure 3.8: Evaluation of the FTS program using the combination of DRE, DRS, DRA and ERGC

9 to 12). We fail again, now to the follower node 3 and let the follower node *steal* the last matching clause ($c3$) that matches with the subgoal $a(X)$. The answer $X = 2$ is found and added to the table space of subgoal $a(X)$ (step 13). The answers for $a(X)$ are then propagated to the context of subgoal $b(X)$ (steps 15 to 20) and a new answer $X = 2$ is found for $b(X)$ at step 19. As the subgoal $b(X)$ do not have any further clauses to be evaluated and it is not a leader call, the fix-point check fails and we start propagating its answers to the context of the pioneer call of subgoal $a(X)$. The first round of evaluation over the SCC is then concluded at step 32. At step 33, we start a second round of evaluation, which will be the last because all the answers

were already found on the first round. This means that the evaluation of the FTS program is concluded in 57 steps, which is again earlier that the previous evaluation that was concluded in 72 steps. Since all the answers were found during the first round of evaluation, the SCC was only evaluated twice. The DRE optimization privileges the execution of program clauses instead of consuming answers from the table space. In particular, as the FTS program has three facts (clauses *c*3, *c*5 and *c*6), the first round of evaluation was in fact very productive. Due to this fact, the total number of evaluation's steps was reduced in 15 steps. In summary, the evaluation of the FTS program with the combination of the DRE, DRS, DRA and ERGC optimizations, is a good example that shows the potential of combining these different optimizations.

Figure 3.9 shows snapshots of the local stack during the evaluation of the first round over the SCC in Fig. 3.8. Analyzing now the general behavior of the local stack, we can observe that it is, in fact, quite different from the previous one. The sinusoidal aspect was replaced by a normal curve and the maximum stack usage is higher than that of the other evaluations (Fig. 3.9 (d)). It is achieved, when the local stack has two generator choice points plus two follower choice points. This is an important advantage, but can also be an important drawback for the DRE optimization, because it makes it more suitable for local stack overflows.

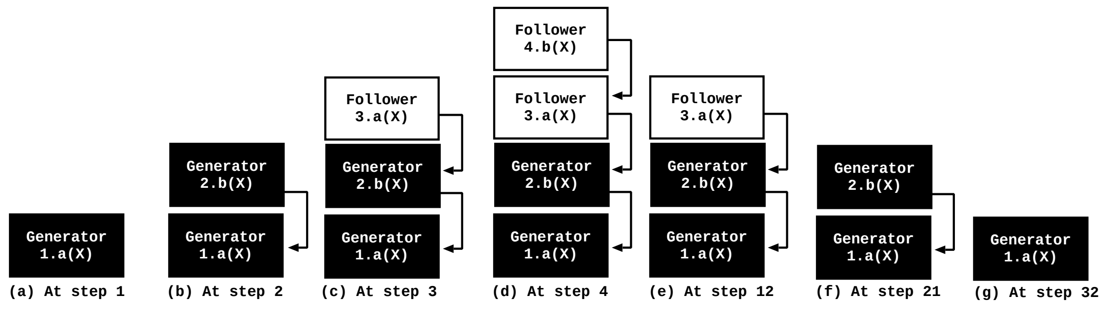


Figure 3.9: Local stack configuration for the evaluation of Fig. 3.8

3.6 Chapter Summary

This chapter revised the most important ideas of the standard linear tabling strategy and optimizations. It began with the description of the standard approach and then followed with the presentation of the four optimizations implemented on top of it. During the chapter, we also analyzed and compared the local stack behavior for the standard evaluation and for the ERGC and DRE optimizations.

Chapter 4

Implementation Details

This chapter describes the main low-level details involved in the implementation of the linear tabling strategies presented in the previous chapter. It describes the organization of the data structures used to support linear tabling and how they interact with each other, and the implementation of the operations used to extend the YapTab system for linear tabled evaluation.

4.1 Compilation of Tabled Predicates

In chapter 2, we briefly introduced how tabling engines change the compilation of logic programs. Here, we will move one step further and discuss in more detail the compilation instructions used to control the evaluation's flow of a tabled logic program. Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me` and `table_trust_me` WAM-like instructions in a similar manner to the generic `try_me/retry_me/trust_me` WAM sequence. The `table_try_me` instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. The `table_retry_me` and `table_trust_me` differ from the generic WAM instructions in that they restore a generator choice point rather than a standard WAM choice point. Tabled predicates defined by a single clause are compiled using the `table_try_single` WAM-like instruction, a specialized version of the `table_try_me` instruction for deterministic tabled calls. As an example, consider YapTab's compiled code for a tabled predicate `t/1` defined by a single clause and for a tabled predicate `t/3` defined by several clauses.

```
% predicate definitions
:- table t/1, t/3.
t(X) :- ...
t(a1,b1,c1) :- ...
t(a1,b2,c2) :- ...
t(a1,b1,c3) :- ...
t(a2,b2,c4) :- ...

% compiled code generated by YapTab for predicate t/1
t1_1:  table_try_single t1_1a
t1_1a: 'WAM code for clause t(X) :- ...'

% compiled code generated by YapTab for predicate t/3
t3_1:  table_try_me t3_2
t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2:  table_retry_me t3_3
t3_2a: 'WAM code for clause t(a1,b2,c2) :- ...'
t3_3:  table_retry_me t3_4
t3_3a: 'WAM code for clause t(a1,b1,c3) :- ...'
t3_4:  table_trust_me
t3_4a: 'WAM code for clause t(a2,b2,c4) :- ...'
```

As `t/1` is a deterministic tabled predicate, the `table_try_single` instruction will be executed for every call to this predicate. On the other hand, `t/3` is a non-deterministic tabled predicate, but some calls to it can be deterministic, i.e., defined by a single matching clause. Consider, for example, the calls `t(a2,X,Y)` and `t(X,Y,c3)`. These two calls are deterministic as each of them matches with a single `t/3` clause, respectively, the 4th and 3rd clause.

YapTab uses the demand-driven indexing mechanism of Yap [31] to dynamically generate `table_try_single` instructions for this kind of deterministic calls. The idea behind it is to generate flexible multi-argument indexing of Prolog clauses during program execution based on actual demand. This feature is implemented for static code, dynamic code and the internal database. All indexing code is generated on demand for all and only for the indices required. This is done by building an indexing tree using similar building blocks to the WAM but it generates indices based on the instantiation of the current goal, and expands indices given different instantiations for the same goal. This powerful optimization allows YapTab to execute some calls to non-deterministic tabled predicates like deterministic tabled predicates. This happens when Yap's indexing scheme finds that for a particular call to a non-deterministic tabled predicate, there is only a single clause that matches the call. Next we show an example illustrating the indexed code generated for a non-deterministic call and for two deterministic calls to the previous `t/3` tabled predicate.

```
% indexed code generated by YapTab for call t(a1,X,Y)
table_try   t3_1a
table_retry t3_2a
table_trust t3_3a

% indexed code generated by YapTab for call t(a2,X,Y)
table_try_single t3_4a

% indexed code generated by YapTab for call t(X,Y,c3)
table_try_single t3_3a
```

The call `t(a1,X,Y)` is non-deterministic as it matches the 1st, 2nd and 3rd clauses of `t/3`, so a `table_try/table_retry/table_trust` sequence is generated. The other two calls, `t(a2,X,Y)` and `t(X,Y,c3)`, are both deterministic as they only match a single `t/3` clause, so a `table_try_single` instruction can be generated.

4.2 Generator and Consumer Nodes

We begin now the presentation of the data structures organization used by YapTab for linear tabling support and the main operations used to manipulate them. As explained previously, tabled nodes are divided into generator or consumer nodes, which correspond respectively to first or repeated calls of a subgoal. The abstract notion of a node is implemented at the engine level as a choice point. Figure 4.1 shows how generator and consumer choice points are implemented in linear tabling.

A generator node is implemented as a standard WAM choice point extended with some extra fields. It's format is depicted in Fig. 4.1(a) and is divided in three sections. The top section contains the usual WAM fields needed to restore the computation on backtracking plus the `cp_sg_fr` field, which is a pointer to the associated subgoal frame (we discuss subgoal frames on the next subsection). The middle section contains

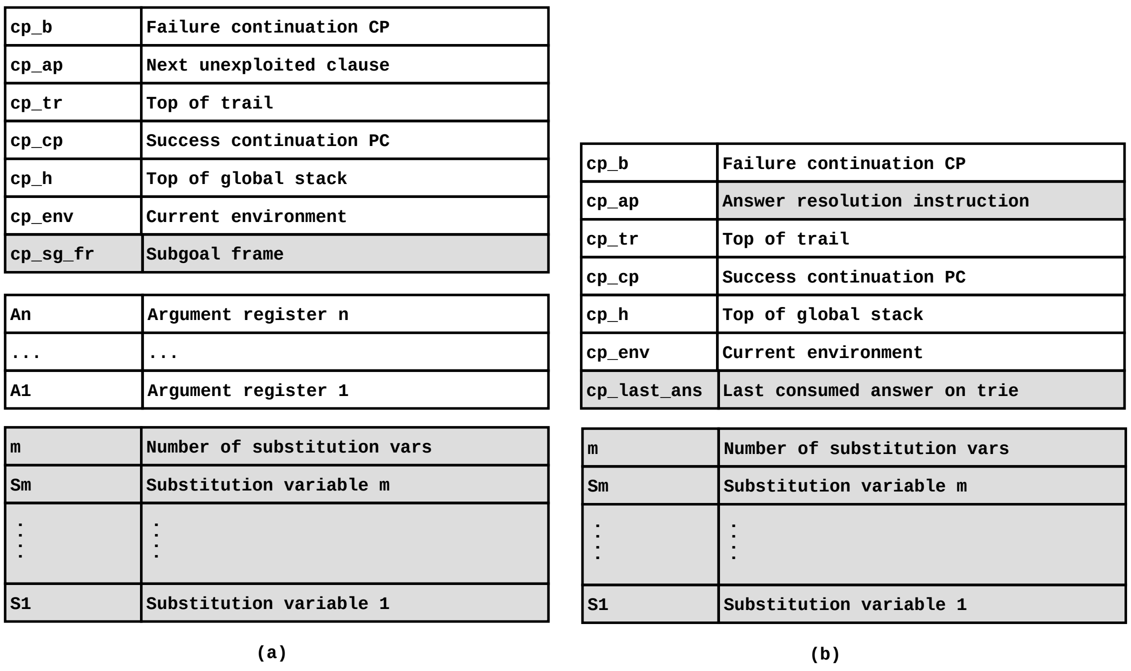


Figure 4.1: Structure of (a) generator and (b) consumer choice points in linear tabling

the argument registers of the subgoal, as usual, and the bottom section contains the *substitution factor* [22], i.e., the set of free variables which exist in the terms of the argument registers. The substitution factor is an optimization that allows the new answer operation to store in the table space only the substitutions for the free variables in the subgoal call.

A consumer node (Fig. 4.1(b)) is similar to a generator node, except that the argument registers disappear and the `cp_sg_fr` pointer is swapped by a `cp_last_ans` pointer, which points directly to the corresponding answer trie structure. Another difference is the fact that the `cp_ap` points to a specific table instruction, that controls how answers are consumed from the table space, instead of pointing to the next unexploited clause as for the generators.

4.3 Subgoal Frames

To implement the table space, YapTab uses *tries* which is regarded as a very efficient way to implement the table space [22]. Tries are trees in which common prefixes are represented only once. Tries provide complete discrimination for terms and permit lookup and insertion to be done in a single pass. Figure 4.2 details the table space organization for the FTS program, which was presented on the previous chapter.

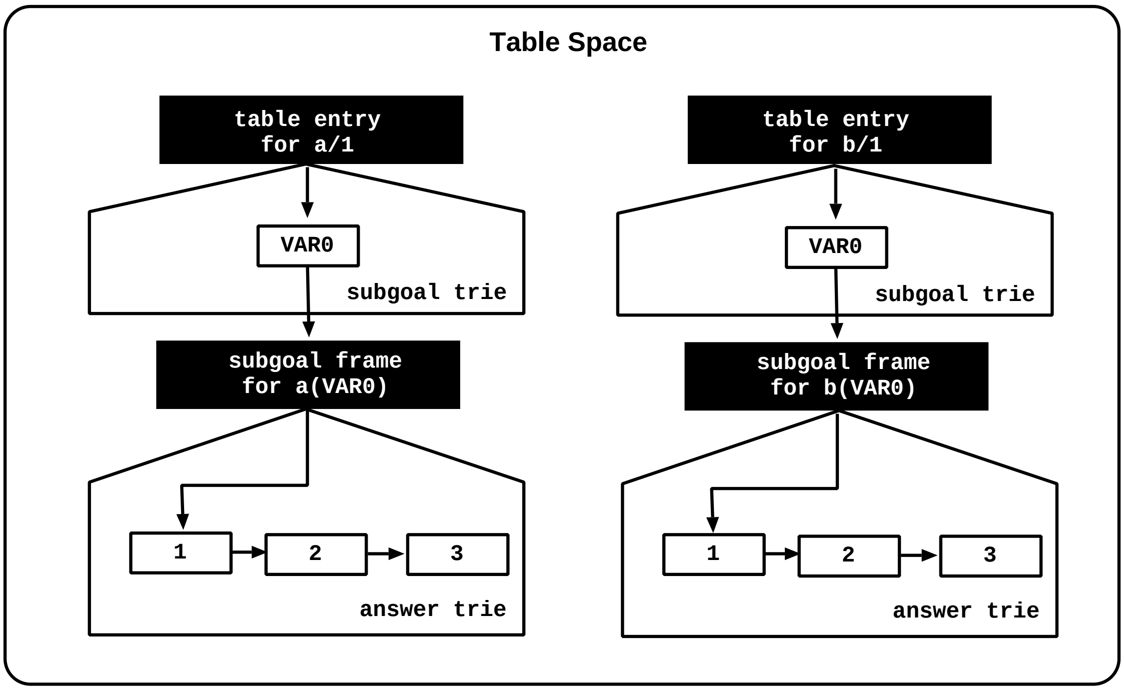


Figure 4.2: Table space organization for the FTS program

YapTab implements tables using two levels of tries: one for the subgoal calls and the other for the computed answers. A tabled predicate accesses the table space through a specific *table entry* data structure. Each different subgoal call is represented as a unique path in the *subgoal trie* and each different answer is represented as a unique path in the *answer trie*.

The subgoal frames not only connect the subgoal with the answers, but they are also a key data structure in the control flow of a tabled computation. Lets start analyzing the basic structure of a subgoal frame used by the standard approach for linear tabling. As we can observe in Fig. 4.3, it includes the following eight fields:

- The **SgFr_dfn** field is the *depth first number* of the subgoal and it is used to detect interdependencies between subgoals. A global variable `CURR_FREE_DFN`, with an initial value of zero, is used to set each different subgoal call with an unique number.
- The **SgFr_is_leader** field is a boolean that defines the leadership of subgoals inside an SCC.

- The **SgFr_gen_cp** field is a back pointer to the corresponding generator choice point.
- The **SgFr_state** field is used to distinguish between three types of possible calls that can happen to a subgoal during evaluation: the first time it is called (`ready`), the follower times (`evaluating`) and when it has been completely evaluated (`complete`).
- The **SgFr_new_answers** field is a boolean that defines if new answers were found during the current evaluation round of the SCC.
- The **SgFr_answer_trie** field points to the top answer trie node and is used to access the answer trie structure to check for/insert new answers.
- The **SgFr_first_answer** field points to the leaf answer trie node of the first available answer.
- The **SgFr_last_answer** field points to the leaf answer trie node of the last available answer.

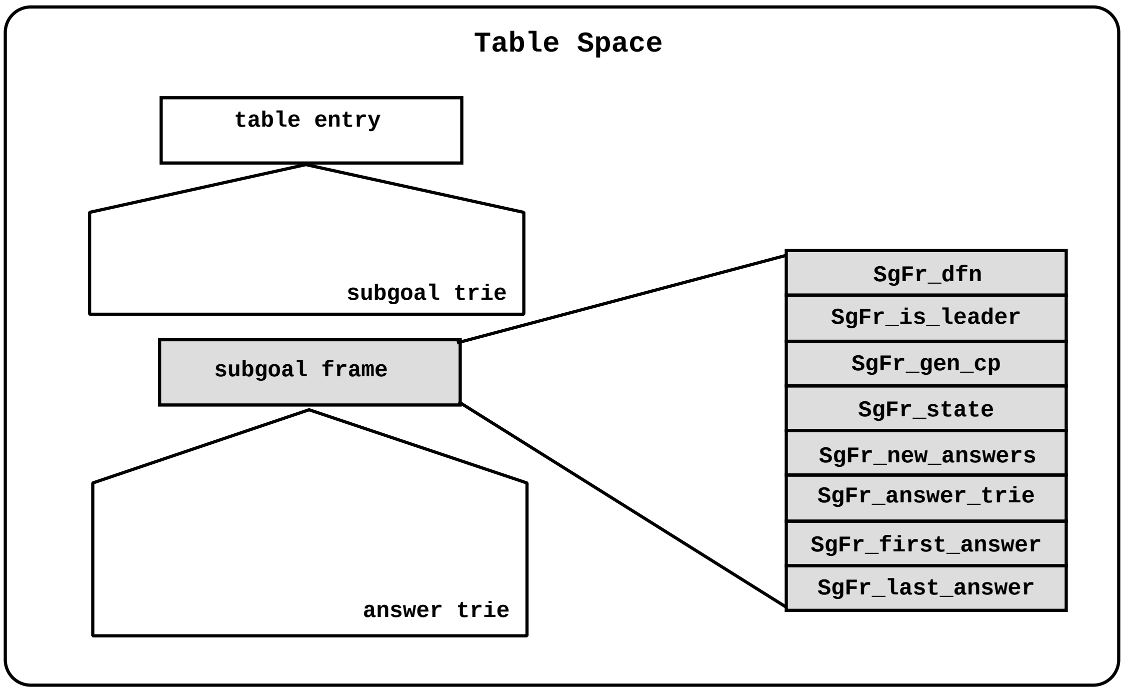


Figure 4.3: Basic structure of a subgoal frame

4.3.1 Looping Structures

Looping structures extend the basic structure of the subgoal frames and they are used to store pointers either to WAM code of alternative clauses or to leaf trie nodes of alternative answers. A looping structure can be seen as groups of buckets of cells. Each bucket has always a last cell which points to the first cell in the next bucket or to the first cell in the first bucket, in the case of the last bucket (see Fig. 4.4). As cells of looping structures are always pointers, the last bit of each cell, is used to mark if the cell is a pointer to an alternative clause/answer or to the first cell in the next bucket.

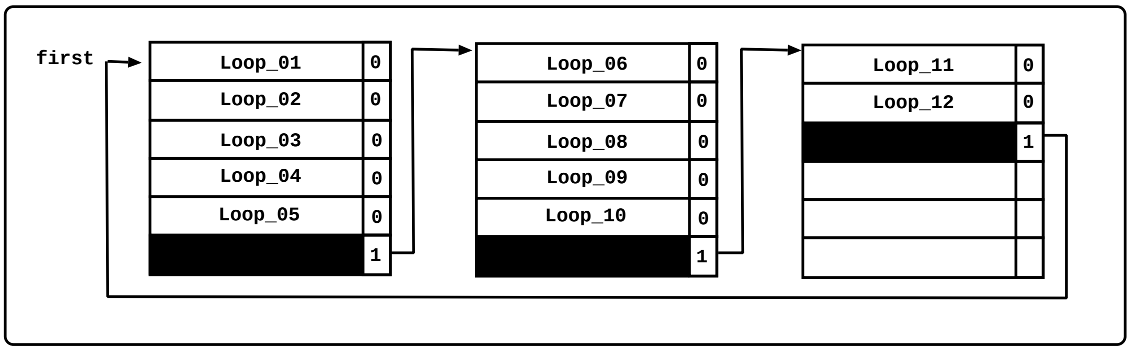


Figure 4.4: A looping structure with three groups of buckets with five cells each

Now, lets observe how the looping structures are used to store alternative clauses. As explained before, the first round of evaluation over an SCC is used to determine dependencies between subgoals and which alternatives lead to repeated computations. Standard linear tabling uses the naive approach of considering that all alternatives must be explored on each round over the SCC, so it adds all alternatives to the looping structure. Using the example of predicate `t/3` from section 4.1, Fig. 4.5 shows how a subgoal frame is extended with a looping structure to store the `t/3` clauses. Since on the following rounds, we only want to execute the alternative's code, the pointer stored in each looping cell points not to the table instruction which starts the alternative, but to its WAM code, avoiding this way the useless execution of tabled instructions. If DRA optimization is active during evaluation, then only the alternatives leading to repeated computations are stored in the looping structures. Control of looping alternatives is provided by two extra fields added to the subgoal frame structure. On each round over the SCC, `SgFr_stop_alt` marks the last alternative to be explored and `SgFr_current_alt` marks the current alternative in evaluation.

A second possible use of the looping structures is to store looping answers when

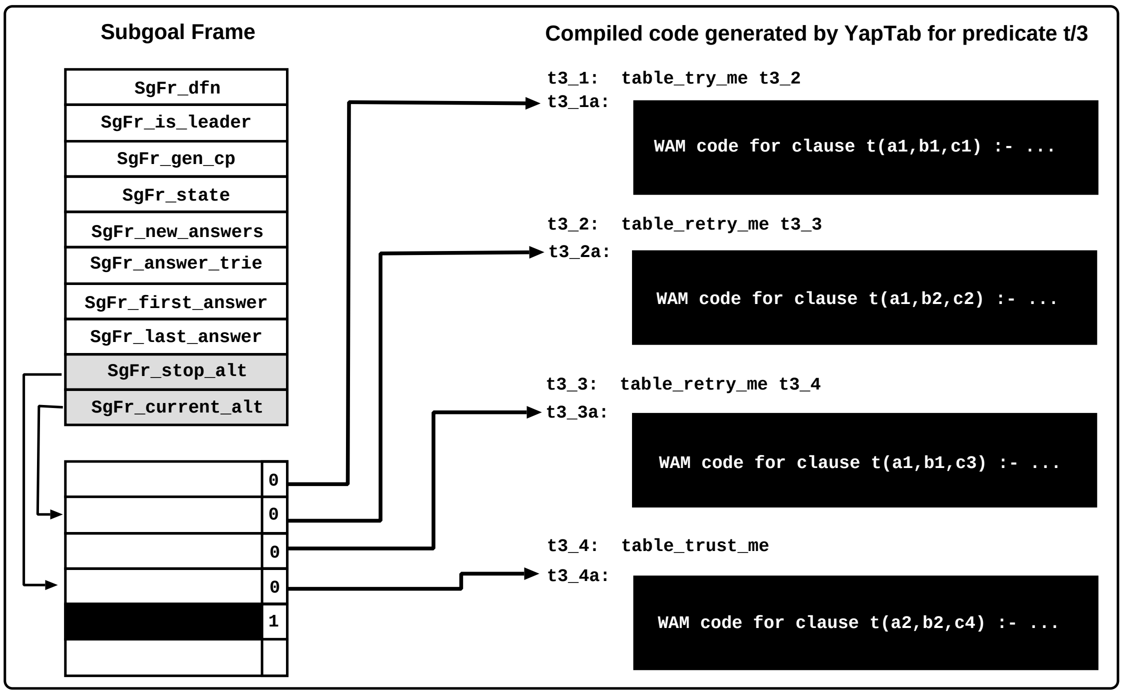

Subgoal Frame
Compiled code generated by YapTab for predicate t/3
SgFr_dfn
SgFr_is_leader
SgFr_gen_cp
SgFr_state
SgFr_new_answers
SgFr_answer_trie
SgFr_first_answer
SgFr_last_answer
SgFr_stop_alt
SgFr_current_alt
0
0
0
0
1
t3_1: table_try_me t3_2
t3_1a:
WAM code for clause t(a1,b1,c1) :- ...
t3_2: table_retry_me t3_3
t3_2a:
WAM code for clause t(a1,b2,c2) :- ...
t3_3: table_retry_me t3_4
t3_3a:
WAM code for clause t(a1,b1,c3) :- ...
t3_4: table_trust_me
t3_4a:
WAM code for clause t(a2,b2,c4) :- ...

Figure 4.5: Using a looping structure to store alternative clauses

the DRS optimization is active. In this case, the subgoal frame structure is then augmented with four extra fields (see Fig. 4.6):

- The `SgFr_new_ans_trie` field marks the first new answer found during the current round of evaluation, if any, and is used by the DRS optimization as the starting position for consuming the new answers.
- The `SgFr_consuming_ans` field is marks the answer found during the current round, which is being consumed, if any.
- The `SgFr_stop_loop_ans` field marks the last looping answer to be consumed on each round.
- The `SgFr_current_loop_ans` field marks the looping answer which is being consumed.

The figure shows a situation where we have nine answers on the answer trie structure. The four first answers (`Ans_01`, `Ans_02`, `Ans_03` and `Ans_04`) were found to be looping answers on the last round of evaluation of the subgoal, so they were added to the

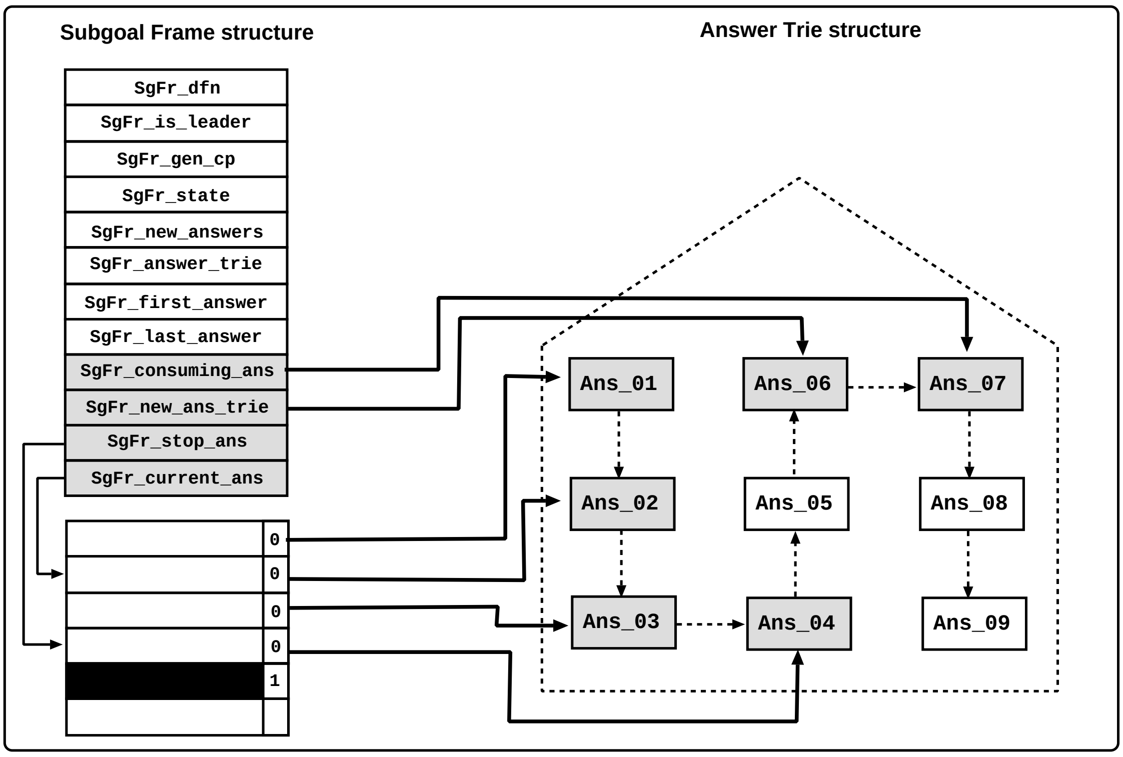


Figure 4.6: Using a looping structure to store alternative answers

looping answers structure. The answers `Ans_05` and `Ans_06`[1] are not looping answers and we are currently consuming the answer `Ans_07`.

4.3.2 DRE Support

In chapter 3, we showed how the DRE optimization changes the execution flow of a tabled evaluation, by allocating a new generator choice point even when a subgoal is a repeated (follower) call.

At the implementation level, whenever this optimization is active, the generator choice points are slightly changed. The `cp_ap` field instead of pointing to the next unexplored alternative, it points to the *fix-point_check* instruction which is responsible to determine if all alternatives were explored and, if not, take the next unexplored alternative. Moreover, the subgoal frame structure is increased with two extra fields. The first is the `SgFr_pioneer` field that will store the generator choice point created by

[1]The answer `Ans_06` was the first new answer found for the subgoal on the current round of evaluation.

the first call to the subgoal. Remember that first (pioneer) calls in DRE evaluation are still where leader calls can complete all the subgoals inside an SCC. The second is the `SgFr_next_alt` field and points to the next unexplored alternative, while the subgoal is in the `evaluating` state (when a subgoal enters the looping state, the alternatives are controlled with the looping structures fields).

Figure 4.7 uses again the compiled code of section 4.1, to illustrate how alternatives are explored using DRE evaluation, in a situation where we are evaluating two subgoal calls to `t(X,Y,Z)`.

As we can observe in the figure, two choice points are allocated, each with its `cp_ap` field pointing to the `fix-point_check` instruction, and the `SgFr_pioneer` field of the corresponding subgoal frame is made to point to the first (pioneer) choice point. If during the evaluation, we had a third call to this subgoal, a new generator choice point would be allocated on the local stack, and the evaluation would start from the alternative pointed by the `SgFr_next_alt` field. Moreover, the `SgFr_next_alt` field would be updated to the next unexplored alternative (`t3_4` in the example).

4.3.3 Subgoal Frame Chains

On the previous chapter, the notions of leader and non-leader subgoals, subgoal dependencies and the process of scheduling the re-evaluation of an SCC only when the leader has new answers, were always presented implicitly. Here, we show the details behind those notions. Besides that, YapTab has important operations, such as garbage collections or local stack overflow recoveries, which need coherence between the choice points on the local stack and the active subgoals in evaluation. This section discusses the three different chains needed for an optimal support of all these features.

Each chain has a global variable that marks the beginning of the chain and each subgoal frame was extended with three fields that are used to follow each chain. The chains are described as follows:

TOP_SG_FR. When YapTab runs out of memory space during evaluation, it starts operations to recover or expand its current space. During these operations, the choice point memory addresses on the local stack are most likely to change and thus all the pointers to these choice points, such as the `SgFr_gen_cp` and the `SgFr_pioneer` fields, must be updated. The `TOP_SG_FR` global variable is used to chain all the subgoal frames that have choice points on the local stack so that the

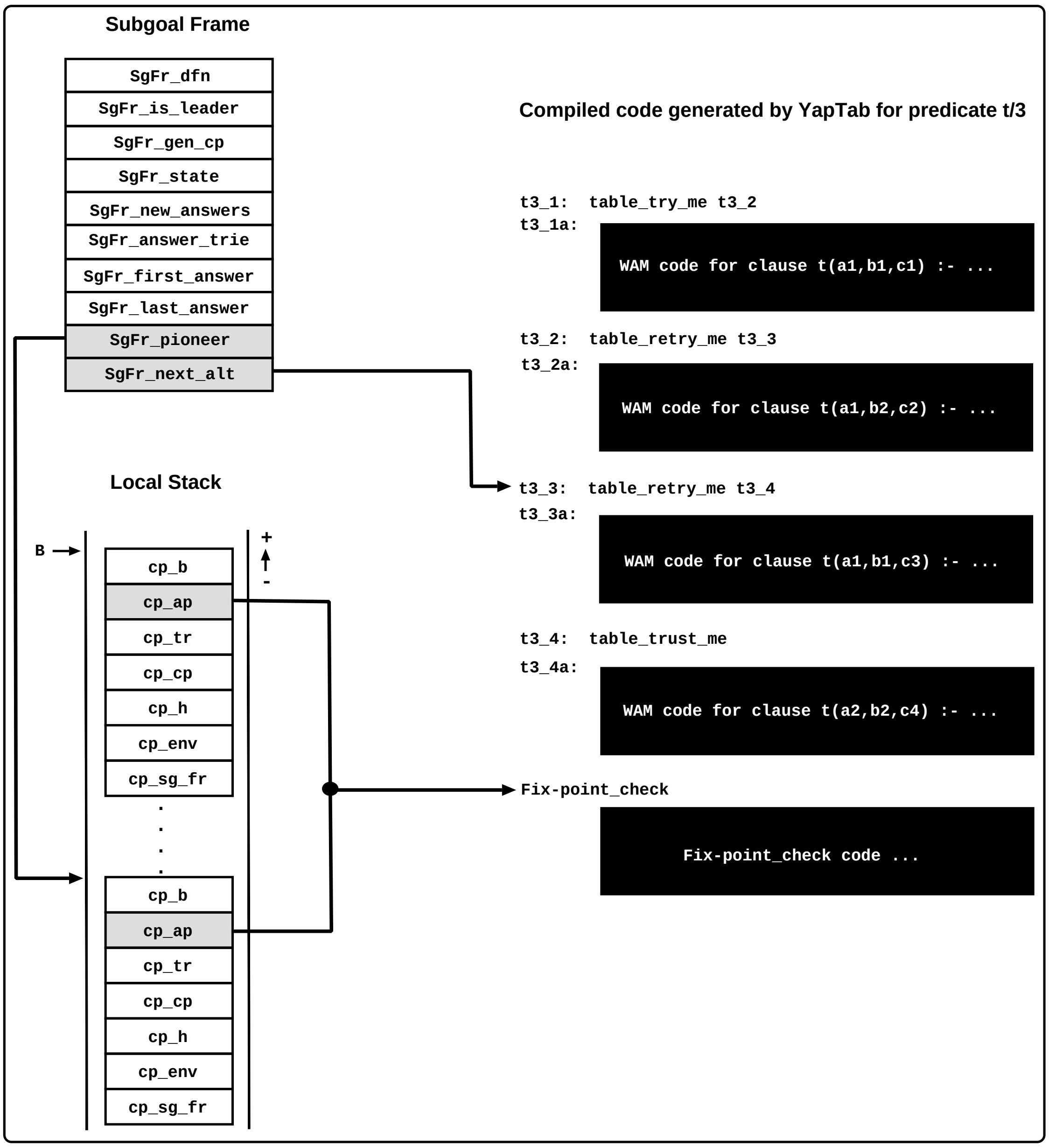


Figure 4.7: Subgoal frame support for DRE optimization

SgFr_gen_cp and SgFr_pioneer fields can be correctly updated. To accomplish that, the subgoal frame structure is then extended with a new SgFr_next field.

TOP_ON_BRANCH. The detection of leaders of an SCC or marking the new answers flag of them can be a very expensive operation. In a worst case scenario, it would mean traversing all subgoal frames in evaluation, for marking a single subgoal frame. The TOP_ON_BRANCH global variable always points to the youngest subgoal frame on the current branch that is in the normal state (i.e., with

`SgFr_state` as `evaluating`) or that is a leader call. When the DRA or DRS optimizations are enabled, this chain is also used to mark the subgoal frame looping alternative clauses or answers, respectively. To implement this chain, the subgoal frame structure is extended with a new `SgFr_next_on_branch` field.

TOP_ON_SCC. This chain is used for the ERGC optimization. The `TOP_ON_SCC` global variable always points to the youngest subgoal frame in evaluation in the current SCC (i.e., all subgoal frames with `SgFr_state` as `evaluating` or `loop_evaluating` are in the chain). It is used by the leader call to traverse the subgoal frames in order to mark them for re-evaluation or as completed. By other words, we use this chain to identify all the subgoal frames inside an SCC, i.e., which were executed at least one time during the current round of evaluation. When a leader marks an SCC for re-evaluation, it uses this chain to set all dependent subgoals with the `loop_ready` state, meaning that whenever they are called again for re-evaluation, they will work again as a first call and start exploring their looping alternatives. Their state is then changed to `loop_evaluating`, which means that on future calls during the current evaluation round, they will again only allocate consumer nodes. To accomplish this, the subgoal frame structure is then extended with a new `SgFr_next_on_scc` field and the `SgFr_state` field is extended with two more states (`loop_ready` and `loop_evaluating`).

To better understand how these chains work, lets consider again the evaluation illustrated on Fig. 3.6 and the structure of subgoal frames presented on the previous sections. In Fig. 4.8, we begin by recalling the evaluation sequence of subgoal calls for the example in Fig. 3.6.

The first subgoal called is $a(X)$, that then calls $b(X)$ on step 2, which calls again $a(Y)$ on step 3. At this step, the subgoal $b(X)$ is marked as non-leader and, at step 9, the fix-point check operation for subgoal $b(X)$ fails because it is not the leader of the SCC. The evaluation then backtracks to the generator node of the subgoal $a(X)$ (which is still under the `evaluating` state) and the subgoal $a(X)$ calls again the subgoal $b(X)$ on step 14. At step 21, the first round of evaluation is completed, and since new answers were found for both subgoals, the fix-point check operation, moves both subgoals to the looping state and schedules a second round of evaluation. The second round finds new answers for subgoal $b(X)$, so a third and last round of evaluation is scheduled on step 48. Finally, at step 75, the fix-point check operation marks both subgoals, $a(X)$ and $b(X)$, as `complete`.

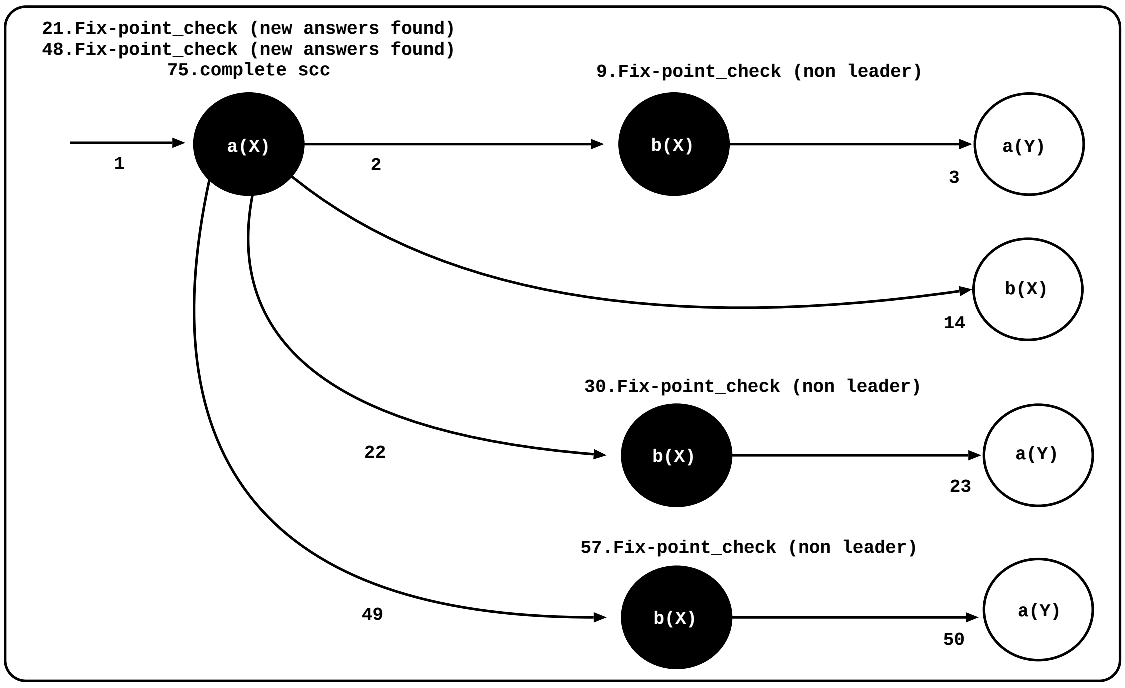

21.Fix-point_check (new answers found)
48.Fix-point_check (new answers found)
75.complete scc
9.Fix-point_check (non leader)
30.Fix-point_check (non leader)
57.Fix-point_check (non leader)
a(X)
b(X)
a(Y)
b(X)
b(X)
a(Y)
b(X)
a(Y)
1
2
3
14
22
23
49
50

Figure 4.8: Subgoal calls sequence for the example in Fig. 3.6

Figure 4.9 helps to understand how the mechanism of the chains works for this example. It shows snapshots that illustrate in more detail what happens with the subgoal frames fields and chains during evaluation. On step 1, the subgoal frame $a(X)$ is initialized. Next, on step 2, the same happens to the subgoal frame $b(X)$, and the `SgFr_next`, `SgFr_next_on_branch` and `SgFr_next_on_scc` fields of $b(X)$ are made to point to $a(X)$ (for lack of space, these fields are not illustrated in the figure).

At step 3, $a(X)$ is called again and starts propagating dependencies. Starting from the `TOP_ON_BRANCH` subgoal frame, it follows the chain and marks the subgoals inside the dependency graph as non-leaders, i.e., the subgoals which have a `SgFr_dfn` value greater than the value of the subgoal that started the propagation of dependencies. In this case $a(X)$ will be kept as leader and $b(X)$ will be marked as non-leader.

At step 5, a new answer is found for $b(X)$ and its `SgFr_new_answers` field is updated to `TRUE`. At step 9, the fix-point check operation fails for $b(X)$ since it is not a leader, thus the choice point of $b(X)$ is popped off from the local stack and the `TOP_ON_BRANCH` and `TOP_SG_FR` are updated to $a(X)$. Moreover, the new answers information in $b(X)$ is passed to $a(X)$ and the `TOP_ON_SCC` is left pointing to $b(X)$.

At step 21, we have another fix-point check operation, but now the computation is at the leader of the SCC. The leader thus follows the `TOP_ON_SCC` chain to mark the state

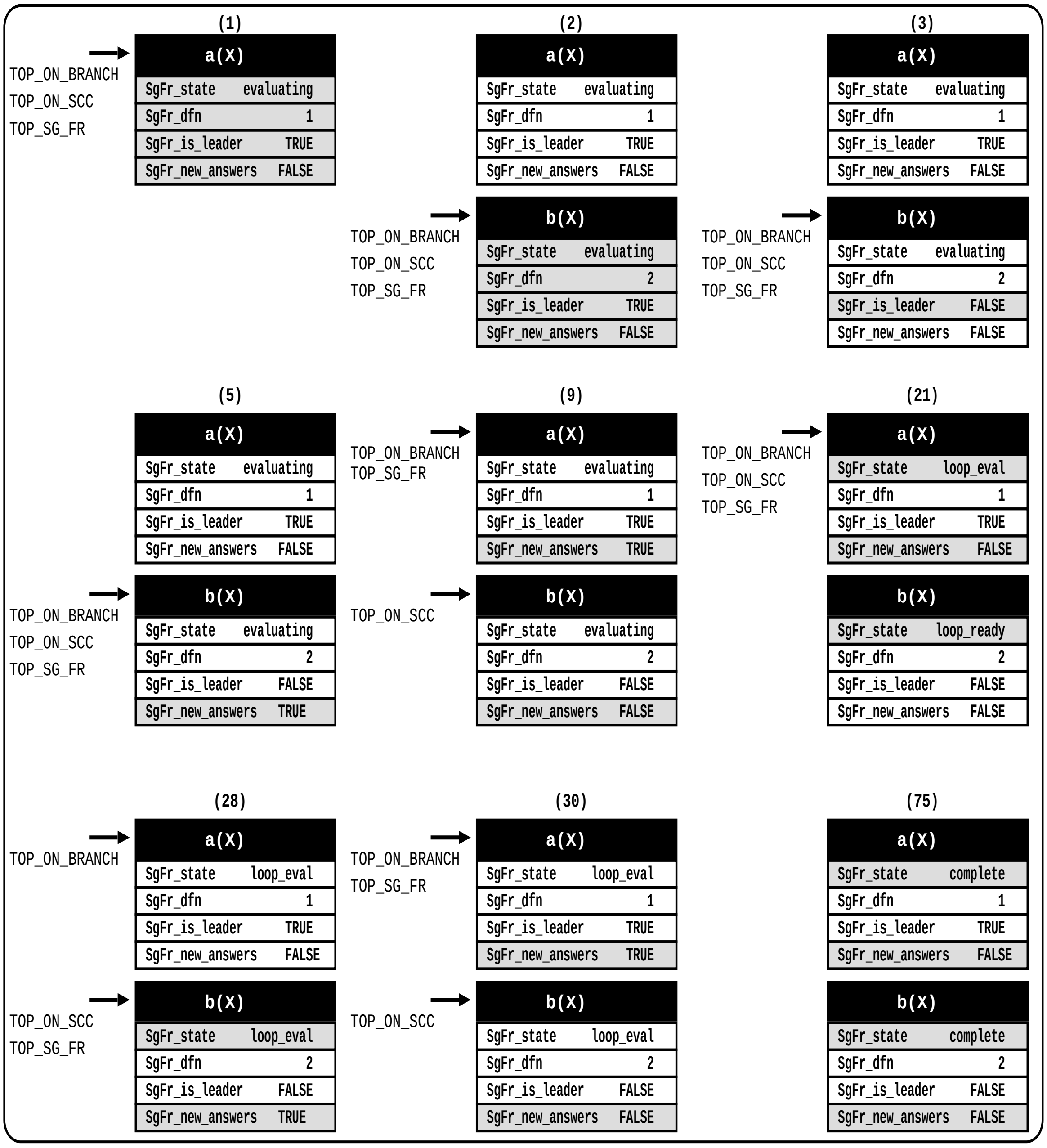


Figure 4.9: Leader detection and dependency propagation for the example in Fig. 3.6

of subgoal $b(X)$ as `loop_ready`. This means that the next time $b(X)$ will appear, it will be the first time for the new round and, thus, a new generator choice point should be allocated. Step 28 shows the `TOP_ON_SCC` and `TOP_SG_FR` global variables, pointing again to subgoal $b(X)$, and the new answers flag marked again as `TRUE`. At step 30, another fix-point check operation fails for $b(X)$ and the new answers flag of $a(X)$ is again marked as `TRUE`, so the SCC is marked for another re-evaluation. Finally, at step 75, both subgoals are marked as `complete` and the evaluation is completed.

4.4 Tabling Instructions

This section introduces the pseudo-code for the main tabling instructions required to support the integration of the linear system and its optimizations on top of YapTab. In order to keep aside from small implementation differences which are not the focus of this thesis, the pseudo-code that we present next for the tabling instructions and the fix-point check operation, abstracts these small details when they are not relevant for the discussion at hand.

4.4.1 Tabling Instructions

We begin with Fig. 4.10 showing the pseudo-code for the `tabled_new_answer()` instruction. Initially, the instruction simply inserts the given answer `ANS` in the answer trie structure for the given subgoal frame `SF` and, if the answer is new, it updates the `SgFr_new_answers` subgoal frame field to `TRUE`. If DRS mode is enable for the subgoal, it also marks the newest answer found during the current round (remember that the `SgFr_new_ans_trie` field is used for it). Otherwise, if the answer `ANS` is repeated, then the instruction simply fails. As we are considering a local scheduling strategy, in any case the instruction fails at the end.

```
tabled_new_answer(answer ANS, subgoal frame SF) {
  if (answer_check_insert(ANS,SF) == TRUE) {             // new answer
    SgFr_new_answers(SF) = TRUE
    if (DRS_mode(SF) && SgFr_new_ans_trie(SF) == NULL)
      SgFr_new_ans_trie(SF) = ANS
  } else                                             // repeated answer
    fail()
  if (local_scheduling_mode(SF))
    fail()
}
```

Figure 4.10: Pseudo-code for the `tabled_new_answer()` operation

Figure 4.11 shows the pseudo-code for the `tabled_call()` operation, which is used for the evaluation of the first program clause which matches the subgoal call. It implements the `table_try_single`, `table_try` and `table_try_me` instructions. New calls to tabled subgoals are inserted into the table space by allocating the necessary data structures (this is the case where the state of `SF` is `ready`). In such case, the tabled call operation then stores a new generator node initializes the given subgoal frame `SF`, which includes updating its state to `evaluating`, and proceeds by executing the current alternative.

On the other hand, if the subgoal call is a repeated call, then the subgoal frame is already in the table space, and three different situations may occur. First, if the call is already evaluated (this is the case where the state of `SF` is `complete`), the operation consumes the available solutions by implementing the *completed table optimization* which executes compiled code directly from the answer trie structure associated with the completed call [22].

Second, if the call is a first call in a re-evaluating round (this is the case where the state of `SF` is `loop_ready`), the operation updates the state of `SF` to `loop_evaluating`, stores a new generator node, and proceeds by re-executing the first looping alternative or the first matching alternative, accordingly to DRA mode be enabled or disable for the subgoal.

Third, if the call is a consumer call (this is the case where the state of `SF` is `evaluating` or `loop_evaluating`), the operation first marks the current branch as a non-leader branch and, if the DRA or DRS mode are enabled, it also marks the current branch as a looping branch. Next, if DRE mode is enabled and there are unexploited alternatives (i.e., there is a backtracking clause for the former call), it stores a follower node and proceeds by executing the next looping alternative or the next matching alternative, accordingly to DRA mode be enabled or disable for the subgoal. Otherwise, it stores a new consumer node and starts consuming the available answers.

To mark the current branch as a non-leader branch, we follow all intermediate generator calls in evaluation up to the generator call for frame `SF` and we mark them as non-leader calls (note that the call at hand defines a new dependency for the current SCC). To mark the current branch as a looping branch, we follow all intermediate generator calls in evaluation up to the generator call for frame `SF` and we mark the alternatives being evaluated or the answers being consumed by each call, respectively, as looping alternatives or looping answers. To accomplish this, we have implemented the `propagate_dependencies()` procedure, which is detailed on Fig. 4.12. This procedure replaces the `mark_current_branch_as_non_leader_branch()` and the

```
tabled_call(subgoal call SC) {
  SF = subgoal_check_insert(SC)      // SF is the subgoal frame for SC
  if (SgFr_state(SF) == ready) {                          // first round
    store_generator_node(SF)
    init_subgoal_frame(SF)
    SgFr_state(SF) = evaluating
    goto execute(current_alternative())
  } else if (SgFr_state(SF) == loop_ready) {    // re-evaluation round
    SgFr_state(SF) = loop_evaluating
    store_generator_node(SF)
    if (DRA_mode(SF))
      goto execute(first_looping_alternative())
    else
      goto execute(first_alternative())
  } else if (SgFr_state(SF) == evaluating or              // first round
             SgFr_state(SF) == loop_evaluating) {//re-evaluation round
    mark_current_branch_as_a_non_leader_banch(SF)
    if (DRA_mode(SF) or DRS_mode(SF))
      mark_current_branch_as_a_looping_branch(SF)
    if (DRE_mode(SF) && has_unexploited_alternatives(SF)) {
      store_follower_node(SF)
      if (DRA_mode(SF) and SgFr_state(SF) == loop_evaluating)
        goto execute(next_looping_alternative())
      else
        goto execute(next_alternative())
    } else {
      store_consumer_node(SF)
      goto consume_all_answers(SF)
    }
  } else if (SgFr_state(SF) == complete)          // already evaluated
    goto completed_table_optimization(SF)
}
```

Figure 4.11: Pseudo-code for the `tabled_call()` operation

`mark_current_branch_as_a_looping_branch()` procedures of Fig. 4.11.

Initially, we use the TOP_ON_BRANCH chain to mark as non-leader all the subgoals inside the dependency graph and, if the DRA or DRS optimizations are enabled, we also use it to mark the looping alternatives or looping answers, respectively. Note that the propagate_dependencies() procedure follows the TOP_ON_BRANCH chain for subgoals with a depth first number higher than the one of the subgoal which is propagating the dependency (SF), i.e., the one leading to a cycle. If DRS mode is enable for a subgoal and it is consuming an answer, the answer is marked as a looping answer[2]. Else, if DRA mode is enable, then the current alternative is marked as a looping alternative. At the end of the procedure, if the DRA mode is enabled, the same is done to mark the looping alternatives for the leader subgoal.

```
propagate_dependencies(subgoal frame SF) {
  subgoal = TOP_ON_BRANCH
  while ((SgFr_dfn(subgoal)  >  SgFr_dfn(SF)) or
    (DRS_mode(subgoal) and SgFr_consuming_ans(subgoal))) {
      SgFr_is_leader(subgoal) = FALSE
      if (DRS_mode(subgoal) and SgFr_consuming_ans(subgoal))
        add_as_looping_answer(subgoal,SgFr_consuming_ans(subgoal))
      else if (DRA_mode(subgoal))
        add_as_looping_alternative(subgoal,SgFr_current_alt(subgoal))
      subgoal = SgFr_next_on_branch(subgoal)
  }
  if (DRA_mode(subgoal))                            // leader subgoal
    add_as_looping_alternative(subgoal,SgFr_current_alt(subgoal))
}
```

Figure 4.12: Pseudo-code for the propagate_dependencies() procedure

Regarding the initialization of subgoal frames mentioned on the tabled_call() operation, Fig. 4.13 shows the pseudo-code for the init_subgoal_frame() procedure. It starts by allocating the looping structures for the alternatives (and for the answers if DRS mode is enabled) and a new answer trie. Then, the SgFr_gen_cp is made to point to the current choice point, the SgFr_dfn field is updated and the CURR_FREE_DFN global variable is increased for the next new subgoal frame, the SgFr_new_answers field is initialized with FALSE, meaning that it has no new answers, and the SgFr_is_leader field is initialized with TRUE, meaning that all subgoals are considered by default to be

[2]It is safe to use the condition "DRS_mode(subgoal) and SgFr_consuming_ans(subgoal)", because the leader of the SCC cannot be using the DRS optimization.

leaders, i.e., without dependencies to other subgoals. The last part of the procedure is used to test if DRA mode is enabled or not. If it is enabled, then the current alternative is stored in the `SgFr_current_alt` field (to be used by the `propagate_dependencies()` procedure), otherwise, it is not enabled, so the current alternative is added to the looping structure.

```
init_subgoal_frame(subgoal frame SF) {
  allocate_looping_structures(SF)
  SgFr_answer_trie(SF)  = allocate_new_answer_trie()
  SgFr_first_answer(SF) = SgFr_last_answer(SF) = NULL
  SgFr_gen_cp(SF)       = B
  SgFr_dfn(SF)          = CURR_FREE_DFN
  CURR_FREE_DFN         = CURR_FREE_DFN + 1
  SgFr_new_answers(SF)  = FALSE
  SgFr_is_leader(SF)    = TRUE
  if (DRA_mode(SF))
    SgFr_current_alt(SF) = current_alternative()
  else
    add_as_looping_alternative(SF,current_alternative())
}
```

Figure 4.13: Pseudo-code for the `init_subgoal_frame()` procedure

Next on Fig. 4.14, we show the pseudo-code for the `store_generator_node()` procedure. Remember that the `store_generator_node()` procedure can be called when the subgoal's state is `ready` or `loop_ready`.

If the subgoal's state is `ready`, we then verify if DRE mode is enabled. If it is the case, we allocate a generator choice point on the local stack, update the `B` register to point to the new choice point, update the `cp_ap` field to point to the `tabled_fix-point_check()` operation, store the current choice point register on the `SgFr_pioneer` field and the next alternative to be evaluated on the `SgFr_next_alt` field. If the DRE mode is not enabled, we simply allocate a generator node on the local stack with the `cp_ap` field pointing to the next available alternative and update the `B` register to point to the new choice point. Then, we proceed by adding the subgoal frame to the `TOP_ON_BRANCH` chain.

On the other hand, if the subgoal's state is `loop_ready`, we allocate a generator choice with the `cp_ap` field pointing to the `tabled_fix-point_check()` operation and, if the

```
store_generator_node(subgoal frame SF) {
  if (SgFr_state(SF) == ready) {
    if (DRE_mode(SF)) {
      B = store_generator_choice_point()
      cp_ap(B) = tabled_fix-point_check()
      SgFr_pioneer(SF) = B
      SgFr_next_alt(SF) = next_alternative()
    } else {                                   // DRE mode not enabled
      B = store_generator_choice_point()
      cp_ap(B) = next_alternative()
    }
    add_to_chain(SF,TOP_ON_BRANCH)
  } else {                                      // state is loop_ready
    B = store_generator_choice_point()
    cp_ap(B) = tabled_fix-point_check()
    if (DRE_mode(SF))
      SgFr_pioneer(SF) = B
  }
  add_to_chain(SF,TOP_SG_FR)
  add_to_chain(SF,TOP_ON_SCC)
}
```

Figure 4.14: Pseudo-code for the `store_generator_node()` procedure

DRE mode is enabled, we store the current `B` register on the `SgFr_pioneer` field. We finish the procedure, by adding the subgoal frame to both `TOP_SG_FR` and `TOP_ON_SCC` chains.

We conclude this subsection, by showing the pseudo-code for the remaining tabling instructions, the `table_retry`, `table_retry_me`, `table_trust` and `table_trust_me` instructions. Fig. 4.15 shows the pseudo-code for the `tabled_retry()` operation that abstracts the instructions `table_retry` and `table_retry_me`. We start the operation by restoring the current choice point (pointed by the `B` register), in order to put the evaluation on the state which was previous to the subgoal call. Again, if DRE optimization is enabled, then we also store the next alternative to be evaluated in the `SgFr_next_alt` field, otherwise, we only restore the current choice point and replace the backtracking alternative for the next alternative.

```
tabled_retry(subgoal frame SF) {
  restore_generator_choice_point(B)
  if (DRE_mode(SF)
    SgFr_next_alt(SF) = next_alternative()
  else
    cp_ap(B) = next_alternative()
  if (DRA_mode(SF))
    SgFr_current_alt(SF) = current_alternative()
  else
    add_as_looping_alternative(SF,current_alternative())
  goto execute(current_alternative())
}
```

Figure 4.15: Pseudo-code for the `tabled_retry()` operation

If DRA optimization is enabled, then the `SgFr_current_alt` field is updated to store the current alternative, for the case of an eventual future dependency cycle. Otherwise, the current alternative is added to the looping structure. At the end, the operation proceeds by executing the current alternative.

Figure 4.16 shows the pseudo-code for the `tabled_trust()` operation that abstracts the instructions `table_trust` and `table_trust_me`. These instructions represent the last program alternative which matches with a subgoal call.

```
tabled_trust(subgoal frame SF) {
  restore_generator_choice_point(B)
  cp_ap(B) = tabled_fix-point_check()
  if (DRE_mode(SF)
    SgFr_next_alt(SF) = NULL
  if (DRA_mode(SF))
    SgFr_current_alt(SF) = current_alternative()
  else
    add_as_looping_alternative(SF,current_alternative())
  goto execute(current_alternative())
}
```

Figure 4.16: Pseudo-code for the `tabled_trust()` operation

We start this operation by restoring the current choice point and by updating the `cp_ap` field to point to the `tabled_fix-point_check` operation. If DRE optimization is enabled, we mark the `SgFr_next` field as `NULL`. We will use this information on the `tabled_fix-point_check()` operation to stop consuming program clauses. This information means also that no more follower nodes will be allocated while the subgoal is in the `evaluating` state. Then, we proceed with the test for inserting the current alternative on the looping structure or not. The instruction ends by executing the current alternative.

4.4.2 Fix-Point Check Operation

This subsection discusses in more detail the fix-point check operation. Remember that with the DRE mode enabled or after exploring the last matching clause for a tabled call or while the subgoal call is in a loop state, we always execute the `tabled_fix-point_check()` operation when backtracking to a generator node. Figure 4.17 shows the pseudo-code for its implementation.

We begin the fix-point check operation by first evaluating the remaining alternatives that match the tabled call and, only if no alternatives exit for the current round, we execute the following code for the `tabled_fix-point_check()` operation. The pseudo-code for the `evaluate_next_alternative()` procedure is presented next on Fig. 4.18. Thus, if no alternatives exist, then we check if the subgoal at hand is a leader call. If it is a leader with new answers found during the current round, we prepare all the subgoals inside the SCC for a new round of evaluation and begin evaluating the first looping alternative (currently pointed by the `SgFr_current_alt` field). If the subgoal is leader but no new answers were found during the current round, then we have reached a fix-point and thus we pop off the generator choice point from the local stack, mark the subgoals in the current SCC as completed, remove their looping structures and remove them from the `TOP_ON_SCC` chain. After that and because we are still only considering the local scheduling strategy, we proceed the evaluation with the completed table optimization. On the next chapter, when we present the batched scheduling strategy, we will observe that the behavior of this operation at this step will be different.

The last part of the operation is related with non leader subgoal calls. In this situation, we pop off the generator choice point, propagate the new answers info to the current leader of the SCC, and start consuming the available answers (due to local scheduling

```
tabled_fix-point_check(subgoal frame SF) {
  evaluate_next_alternative(SF)
  if (SgFr_is_leader(SF)) {
    if (SgFr_new_answers(SF)) {     // prepare the SCC for a new round
      SgFr_new_answers(SF) = FALSE
      SgFr_state(SF) = loop_evaluating
      subgoal = TOP_ON_SCC
      while (subgoal != SF) {
        SgFr_state(subgoal) = loop_ready
        remove_from_chain(subgoal,TOP_ON_SCC)
        subgoal = SgFr_next_on_scc(subgoal)
      }
      goto execute(first_looping_alternative())
    } else {                              // leader without new answers
      pop_generator_choice_point(SF)
      subgoal = TOP_ON_SCC
      while (subgoal != SgFr_next_on_SCC(SF)) {
        SgFr_state(subgoal) = complete
        free_looping_structures(subgoal)
        remove_from_chain(subgoal,TOP_ON_SCC)
        subgoal = SgFr_next_on_scc(subgoal)
      }
      goto completed_table_optimization(SF)        // local scheduling
  } else {                                         // not a leader call
    pop_generator_choice_point(SF)
    if (SgFr_new_answers(SF))            // propagate new answers info
      SgFr_new_answers(current_leader(SF)) = TRUE
    SgFr_new_answers(SF) = FALSE            // reset new answers info
    if (DRS_mode(SF))                              // local scheduling
      goto consume_looping_answers_and_answers_in_current_round(SF)
    else
      goto consume_all_answers(SF)
  }
}
```

Figure 4.17: Pseudo-code for the tabled_fix-point_check() operation

strategy). If DRS mode is enabled, we will only consume the looping answers and the answers found during the current round[3], otherwise we consume all the answers.

The implementation of the `evaluate_next_alternative()` procedure is shown next on Fig. 4.18. We begin by supporting the extra control needed by the DRE optimization where we check if all the program clauses that match the tabled call were already evaluated and, if they were not, we proceed by executing the next available alternative. Otherwise, we check if this is the first time (cases where the state of the subgoal is `evaluating`) or a former time (cases where the state of the subgoal is `loop_evaluating`) that the `tabled_fix-point_check()` operation is being executed for the tabled call at hand.

```
evaluate_next_alternative(subgoal frame SF) {
  if (DRE_mode(SF) and SgFr_next_alt(SF))
    goto execute(SgFr_next_alt(SF))
  if (SgFr_state(SF) == evaluating) {
    if (DRE_mode(SF) and SgFr_pioneer(SF) != B) {     // follower node
      pop_follower_choice_point()
      goto consume_all_answers(SF)
    }
    SgFr_state(SF) = loop_evaluating           // move to a loop state
    if (SgFr_is_leader(SF) == FALSE)
      remove_from_chain(SF,TOP_ON_BRANCH)
    SgFr_current_alt(SF) = first_looping_alternative()
    SgFr_stop_alt(SF)    = first_looping_alternative()
  } else {
    SgFr_current_alt(SF) = next_looping_alternative()
    if (SgFr_current_alt(SF) != SgFr_stop_alt(SF))
      goto execute(SgFr_current_alt(SF))
  }
}
```

Figure 4.18: Pseudo-code for the `evaluate_next_alternative()` procedure

For first time situations, we check again if DRE mode is enabled and if the tabled call is a follower call. If it is the case, we pop off the choice point from the local stack

[3]We begin the procedure by consuming the looping answers. Then, we add the subgoal's frame to the `TOP_ON_BRANCH` chain and schedule all its new answers found on the current round for evaluation. After consuming all the answers, we remove the subgoal's frame from the `TOP_ON_BRANCH` chain.

and consume the subgoal's answers. Otherwise, if DRE mode is not enabled or if the call is a pioneer, we move the subgoal's state to a looping state (`loop_evaluating`), remove the subgoal frame from the `TOP_ON_BRANCH` chain if it is a non-leader call[4] and update the `SgFr_current_alt` and `SgFr_stop_alt` fields to point to the first looping alternative. For former time situations, we simply update the `SgFr_current_alt` field to point to the next looping alternative and execute the next unexploited alternative for the subgoal, if the current one is not the last.

We conclude the description of the `tabled_fix-point_check()` operation with the presentation of the `free_looping_structures()` procedure through the Fig. 4.19. This procedure is used to remove the looping structures for alternatives or/and for answers. Notice that even when DRA is not enabled, the looping structures are used to store alternatives and that in such cases, they store all the looping and non-looping alternatives.

To implement this procedure, we use three abstract pointer fields. The `fst_bkt` is used to mark the first bucket, the `curr_bkt` field is used to mark the bucket which will be removed and the `next_bkt` is used to mark the next bucket to be removed. The `next_bkt` field must be updated before cleaning the `curr_bkt` bucket, otherwise we would not be able to jump to the next bucket. So, we start by storing the first cell of the bucket, which stores the first looping alternative, on the `fst_bkt` and `curr_bkt` pointers. Then we proceed with a cycle to clean all the buckets. For each bucket, the `free_bucket()` procedure frees the memory space and updates the looping alternative fields of the subgoal's frame to `NULL`.

After cleaning the looping alternatives buckets, we check if DRS mode is enabled and, if so, we use the same procedure to clean the looping answers buckets.

4.5 Chapter Summary

This chapter described the main implementation details to support linear tabling in YapTab. We described the data structures for supporting the different linear tabling optimizations, we discussed the leader detection algorithm and we presented the details

[4]As an optimization, when a non-leader call moves to the looping state, it can be removed from the `TOP_ON_BRANCH` chain because there is no point in keeping it there. This is the reason why, if later we execute the `propagate_dependencies()` procedure for the call at hand, we need to follow the subgoal frames in the `TOP_ON_BRANCH` chain up to the first subgoal frame with a smaller `SgFr_dfn` value, as described on the pseudo-code for the `propagate_dependencies()` procedure on Fig. 4.12.

```
free_looping_structures(subgoal frame SF) {
  fst_bkt  = get_first_cell_of_bucket(first_looping_alternative())
  curr_bkt = fst_bkt
  do {
    next_bkt = get_first_cell_of_next_bucket(curr_bkt)
    free_bucket(curr_bkt)
    curr_bkt = next_bkt
  } while (curr_bkt != fst_bkt)
  if (DRS_mode(SF)) {
    fst_bkt  = get_first_cell_of_bucket(first_looping_answer())
    curr_bkt = fst_bkt
    do {
      next_bkt = get_first_cell_of_next_bucket(curr_bkt)
      free_bucket(curr_bkt)
      curr_bkt = next_bkt
    } while (curr_bkt != fst_bkt)
  }
}
```

Figure 4.19: Pseudo-code for the `free_looping_structures()` procedure

involved in the execution control of the main linear tabling operations.

Chapter 5

Batched Scheduling

This chapter presents the key ideas of the batched scheduling strategy and all the consequent changes made to the main operations already created to support the local scheduling strategy.

5.1 Key Ideas

Batched scheduling is an alternative strategy that can be used for the evaluation of tabled logic programs. Its importance was recognized when the SLG-WAM, which is the tabling suspension-based mechanism of the XSB Prolog system, started using it as the default strategy (for versions 1.5 and higher of XSB) [11].

The batched scheduling strategy takes its name because it tries to minimize the need to move around the search tree by *batching* the return of answers. When new answers are found, they are added to the table space and the evaluation continues, instead of failing as for the local scheduling strategy. Therefore, the subgoals do not need to consume answers after the fix-point check operation[1]. However, as we can observe through the evaluation shown on Fig. 5.1, for linear tabling, the consumption of answers is still necessary. Since the `tabled_new_answer` operation fails when repeated answers are found in a new re-evaluation round, making each answer to be consumed only once, this may not be sufficient to assure the completeness of an SCC. So, the consumption of answers is still necessary before re-evaluating a tabled subgoal. Figure 5.1 shows an

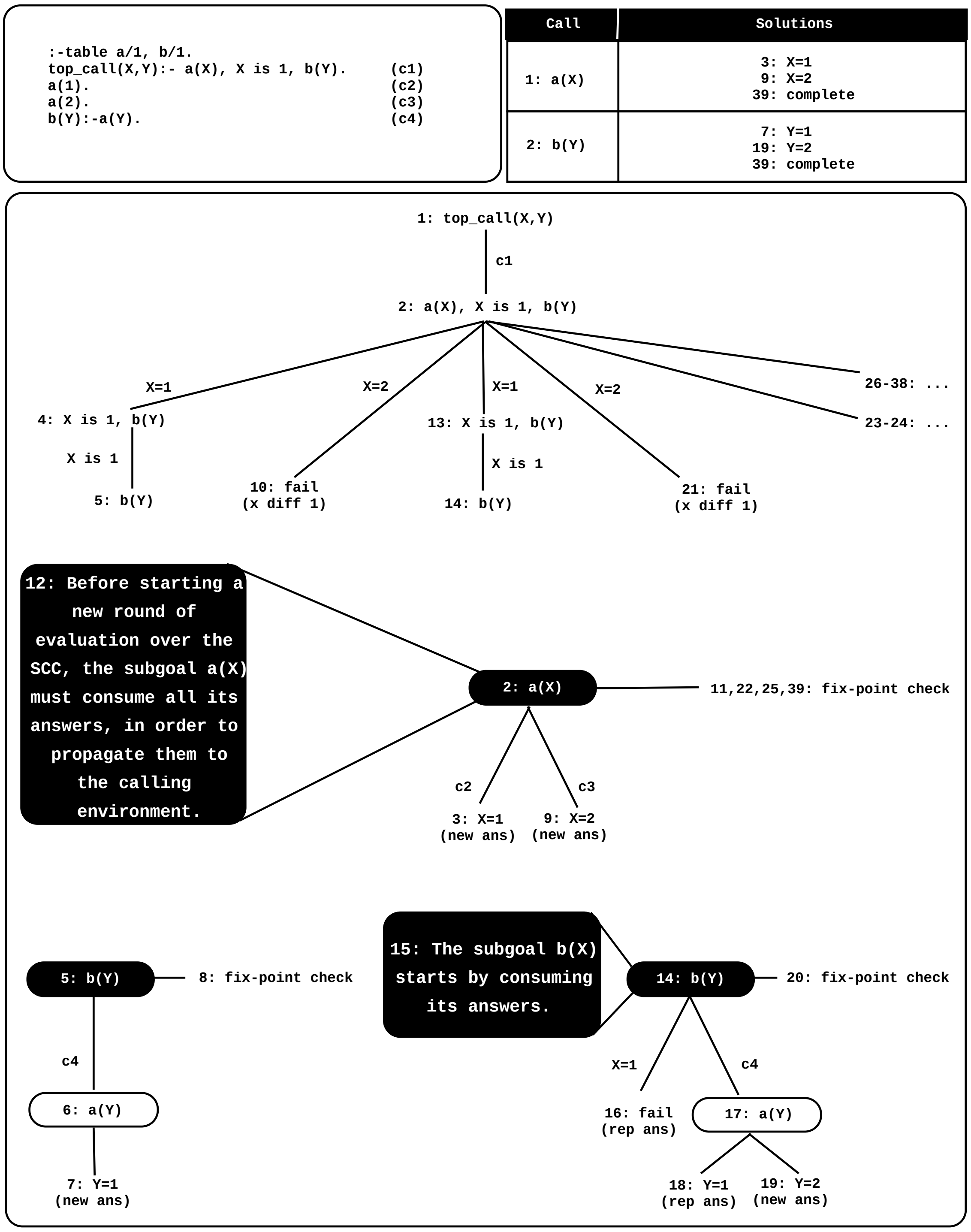
[1]Recall that the DRS optimization is used when the non-leader generator subgoal calls are consuming answers after the fix-point check operation. As a consequence, the application of the DRS optimization is useless on this strategy, so it was not implemented on our system.

example that illustrates the propagation of answers during an evaluation with batched scheduling. The example, which is a variant of the FTS program, has four clauses and two tabled predicates. The goal of the program is to find all the tuples which satisfy the top query call. The solution set of the problem is $\{(1,1),(1,2)\}$.

At step 1, we start evaluating the `top_call(X,Y)` query by executing the clause $c1$, which leads to the subgoal call $a(X)$, and, at the step 2, we allocate a generator node and start evaluating the clause $c2$. At step 3, the new answer $X = 1$ is found and added to the table space of $a(X)$, but now on the `tabled_new_answer` operation we proceed with the evaluation. Next, at step 4, the test `X is 1` succeeds and on the continuation we call the subgoal $b(Y)$ (step 5). We allocate a generator node for $b(Y)$ and start evaluating the clause $c4$, which leads to a repeated call to $a(Y)$ (step 6). At this step, we allocate a consumer node for $a(Y)$ and consume the answer $Y = 1$, leading to a first solution $(X = 1, Y = 1)$ for the top query goal. In the continuation, the evaluation backtracks to node 6 but as no more answers exist to be consumed, it backtracks to node 5. Node 5 has explored all the matching clauses and because $b(Y)$ is not a leader call, it depends on $a(Y)$, we backtrack again and the evaluation reaches node 2. At this node we evaluate the second matching clause $c3$, and find the answer $X = 2$ for the subgoal $a(X)$ (step 9), but now the test `X is 1` fails (step 10), so we backtrack again to node 2. At this point, we perform a fix-point check operation and decide to start a new re-evaluation round.

Suppose now that we would not consume answers at this node before starting the new round of evaluation. Notice that, the answers $X = 1$ and $X = 2$ are already on the table space of $a(X)$, so the re-evaluation of clauses $c2$ and $c3$, leading to those answers, will be blocked by the `tabled_new_answer` operation, as they are repeated answers. This means that no new answer will be found on the new round and the evaluation of the SCC would finish prematurely. In other words, because the subgoal $b(Y)$ depends on subgoal $a(Y)$, the answer $Y = 2$ will not be propagated to the context of the subgoal $b(Y)$, and thus the solution $(X = 1, Y = 2)$ which belong to the solution set of the problem would not be found.

Returning to the evaluation, at the node 2, we thus start by consuming the answers available for subgoal $a(X)$, starting by the answer $X = 1$, and the evaluation reaches again $b(Y)$ (step 14). At this node, we start also by consuming the answers on the table space of $b(X)$ and only afterwards we explore the clause $c4$. This leads to the consumer node 17, which this time propagates the answer $Y = 2$ to the subgoal $b(Y)$ (step 19) and the solution $(X = 1, Y = 2)$ is found. At step 20, we fail the evaluation, because we do not have any more answers/clauses to evaluate, and so we backtrack



```
:-table a/1, b/1.
top_call(X,Y):- a(X), X is 1, b(Y).      (c1)
a(1).                                     (c2)
a(2).                                     (c3)
b(Y):-a(Y).                               (c4)
```

Call	Solutions
1: a(X)	3: X=1 9: X=2 39: complete
2: b(Y)	7: Y=1 19: Y=2 39: complete

Figure 5.1: Propagation of answers on a tabled evaluation using batched scheduling

again to node 2. At step 21, we evaluate the answer $X = 2$, but the test `X is 1` fails, thus we backtrack one more time to node 2. Now we start evaluating the program clauses $c2$ and $c3$, but it does not lead to any further developments on the table space (steps 23 and 24). As the previous evaluations shown on chapter 3, this evaluation finishes with a new round of evaluation over the SCC without new answers (steps 26 to 38), and at step 39, both subgoals $a(X)$ and $b(X)$ are marked as complete.

In summary, the key differences between local and batched scheduling are: (i) batched scheduling does not fail when it finds a new answer as local scheduling does, and (ii) the consumption of the answers on the table space is always made before starting a new round of evaluation of the program clauses, instead of consuming them after the fix-point check operation as with local scheduling.

5.2 Implementation Details

Extending our system to support batched scheduling involves making slight changes to the structures and operations presented on the previous chapter for local scheduling. The first operation that was changed was the `tabled_new_answer()` as shown on Fig. 5.2.

```
tabled_new_answer(answer ANS, subgoal frame SF) {
  if (answer_check_insert(ANS,SF) == TRUE) {              // new answer
    SgFr_new_answers(SF) = TRUE
    if (DRS_mode(SF) && SgFr_new_ans_trie(SF) == NULL)
      SgFr_new_ans_trie(SF) = ANS
  } else                                             // repeated answer
    fail()
  if (local_scheduling_mode(SF))
    fail()
  else                                             // batched scheduling
    proceed()
}
```

Figure 5.2: Pseudo-code for the `tabled_new_answer()` operation with support for batched scheduling

The first part of the operation, where we check for new or repeated answers, remains unchanged. The difference occurs on the second part, where we now check for the scheduling strategy (local or batched). If the scheduling strategy is local, then we continue to fail the evaluation. Otherwise, the strategy is batched, so we adjust the execution's environment and proceed with the evaluation.

The support for the consumption of answers before executing the program clauses required two steps. The first step was to extend the subgoal's frame structure with a new field, which we called `SgFr_batched_ans`. This field is used as a state flag to mark when a subgoal is or is not consuming batched answers, and when the subgoal is consuming answers, it points to the current answer being consumed.

The second step was to support the answer consumption on non-leader and leader subgoals, which involved changing the `tabled_call()` and the `tabled_fix-point_check()` operations, respectively. Figure 5.3 shows the changes made to the `tabled_call()` operation. The three dots represent parts of the pseudo code which are identical to the ones presented previously on Fig. 4.11.

For the support of the consumption of batched answers on non-leader subgoals, we had to change the behavior of the evaluation when the subgoal is called with the `looping_ready` state. As before, we start by updating the state of the subgoal to `looping_evaluating` and by storing a generator node (this is common to both strategies). But now, if it is in batched scheduling mode and before starting the evaluation of the alternatives, we update the batched answer field of the subgoal to its first answer. Then, we execute the `cons_all_bat_ans_and_execute()` procedure in order to consume all batched answers, and execute the alternative procedure. This procedure, turns the generator choice point of the call at hand into a consumer choice point, consumes all batched answers of the subgoal (starting from the first answer) and, when no more answers are to be consumed, the procedure turns the consumer choice point into the initial generator and starts the evaluation of the first alternative.

The other operation changed was the `tabled_fix-point_check()` operation. Figure 5.4 shows the changes made in order to add the support for consuming the batched answers on leader subgoals. We start the procedure before, by first consuming the available alternatives and then we check if the subgoal is leader or not. If the subgoal is leader has new answers and is being evaluated in batched scheduling, before executing the first looping alternative, we update again the batched answer field of the subgoal to its first answer. Then, we execute the `cons_all_bat_ans_and_execute()` procedure in order to consume all batched answers and execute the first alternative procedure. Again, this means that we temporally change the generator choice point into a consumer, consume all batched answers and, when all answers are exhausted, we turn the consumer into a generator and execute the first alternative. If the subgoal is leader but do not have answers, then we complete the SCC and fail the evaluation.

```
tabled_call(subgoal call SC) {
  SF = subgoal_check_insert(SC)      // SF is the subgoal frame for SC
  if (SgFr_state(SF) == ready) {                          // first round
    . . .
  } else if (SgFr_state(SF) == loop_ready) {    // re-evaluation round
    SgFr_state(SF) = loop_evaluating
    store_generator_node(SF)
    if (local_scheduling_mode(SF)){
      if (DRA_mode(SF))
        goto execute(first_looping_alternative())
      else
        goto execute(first_alternative())
    } else {                                     // batched scheduling
      SgFr_batched_ans(SF) = SgFr_first_answer(SF)
      if (DRA_mode(SF))
        goto cons_all_bat_ans_and_execute(first_looping_alternative())
      else
        goto cons_all_bat_ans_and_execute(first_alternative())
    }
  } else if (SgFr_state(SF) == evaluating or            // first round
             SgFr_state(SF) == loop_evaluating) {//re-evaluation round
    . . .
  } else if (SgFr_state(SF) == complete)          // already evaluated
    . . .
}
```

Figure 5.3: Pseudo-code for the `tabled_call()` operation with support for batched scheduling

Notice that with local scheduling, at this point we would consume all the subgoal's answers.

On the other hand, if the subgoal is not leader, after popping off the generator choice point from the local stack and propagating the new answers to the leader of the SCC, with batched scheduling we simply fail the evaluation. Suppose now that new answers were found for the subgoal, would this mean that we might lose solutions inside the SCC? No, because the new answers have been already propagated when they were

```
tabled_fix-point_check(subgoal frame SF) {
  evaluate_next_alternative(SF)
  if (SgFr_is_leader(SF)) {
    if (SgFr_new_answers(SF)) {
      . . .
      if (local_scheduling_mode(SF))
        goto execute(first_looping_alternative())
      else {                                        // batched scheduling
        SgFr_batched_ans(SF) = SgFr_first_answer(SF)
        goto cons_all_bat_ans_and_execute(first_looping_alternative())
      }
    } else {                               // leader without new answers
      . . .
      if (local_scheduling_mode(SF))
        goto completed_table_optimization(SF)
      else                                          // batched scheduling
        fail()
  } else {                                          // not a leader call
    . . .
    if (local_scheduling_mode(SF))
      if (DRS_mode(SF))
        goto consume_looping_answers_and_answers_in_current_round(SF)
      else
        goto consume_all_answers(SF)
    else                                            // batched scheduling
      fail()
  }
}
```

Figure 5.4: Pseudo-code for the `tabled_fix-point_check()` operation with support for batched scheduling

found. Moreover, since the new answers flag of the subgoal is propagated to the leader of the SCC, the leader will mark the SCC for a new round evaluation. In this new round, the subgoal will be called again, and so it will start by consuming all its answers, including the new ones.

5.3 Support for the DRE Optimization

The support for DRE optimization with batched scheduling was one of the most challenging problems that we have faced in order to support batched scheduling, because besides controlling the batched answers on leader and non leader subgoals, with DRE optimization, we have the extra complexity of dealing with pioneer and follower nodes. For the same subgoal, we can have one pioneer node and several followers nodes executing at the same time, in which some of those nodes might be consuming batched answers and others not. As each node is independent from the others and a follower node can have different positions on different evaluation rounds over the SCC (making it hard to identify uniquely each node), the subgoal frame of the subgoal can not be used to store the batched answers for the pioneer and for his follower nodes. In reality, the solution to the problem has in fact far more simple than the problem itself, because all the tools were already created.

Figure 5.5 shows the general picture of the solution. It is divided into three areas, the local stack, with the `B` register pointing to the current choice point, the `TOP_SG_FR` chain, with the `TOP_SG_FR` pointing to the current subgoal in use and the answer trie structure. The figure simulates a situation where, for the same subgoal, we have one pioneer and two followers nodes, nine answers on the answer trie structure and all the three nodes are consuming batched answers. The pioneer is consuming the answer `Ans_09`, the first follower is consuming the answer `Ans_02` and the last follower (which is the topmost) is consuming the answer `Ans_06`.

The solution for the consumption of batched answers was to use the `SgFr_batched_ans` field, which controls the batched answer that is being consumed, on the subgoal frames that were previously used to maintain the coherence between the subgoals in evaluation and the generator choice points on the local stack. Thus, now each subgoal frame in the `TOP_SG_FR` chain has the information about the its generator choice point, the information about the node, if it is a pioneer or a follower and the information about the batched answer which it is consuming, if any. This solution to the consumption of batched answers allows each node to work independently from the others.

At each node, either being a pioneer or a follower, we always have all the necessary information available for the evaluation of the subgoal. If we want to consume batched answers, we use the `TOP_SG_FR` pointer, which always points to the current subgoal in evaluation. If we want to access all the remaining information about the subgoal, then we can go directly to the subgoal's frame, by consulting the `cp_sg_fr` field of the top

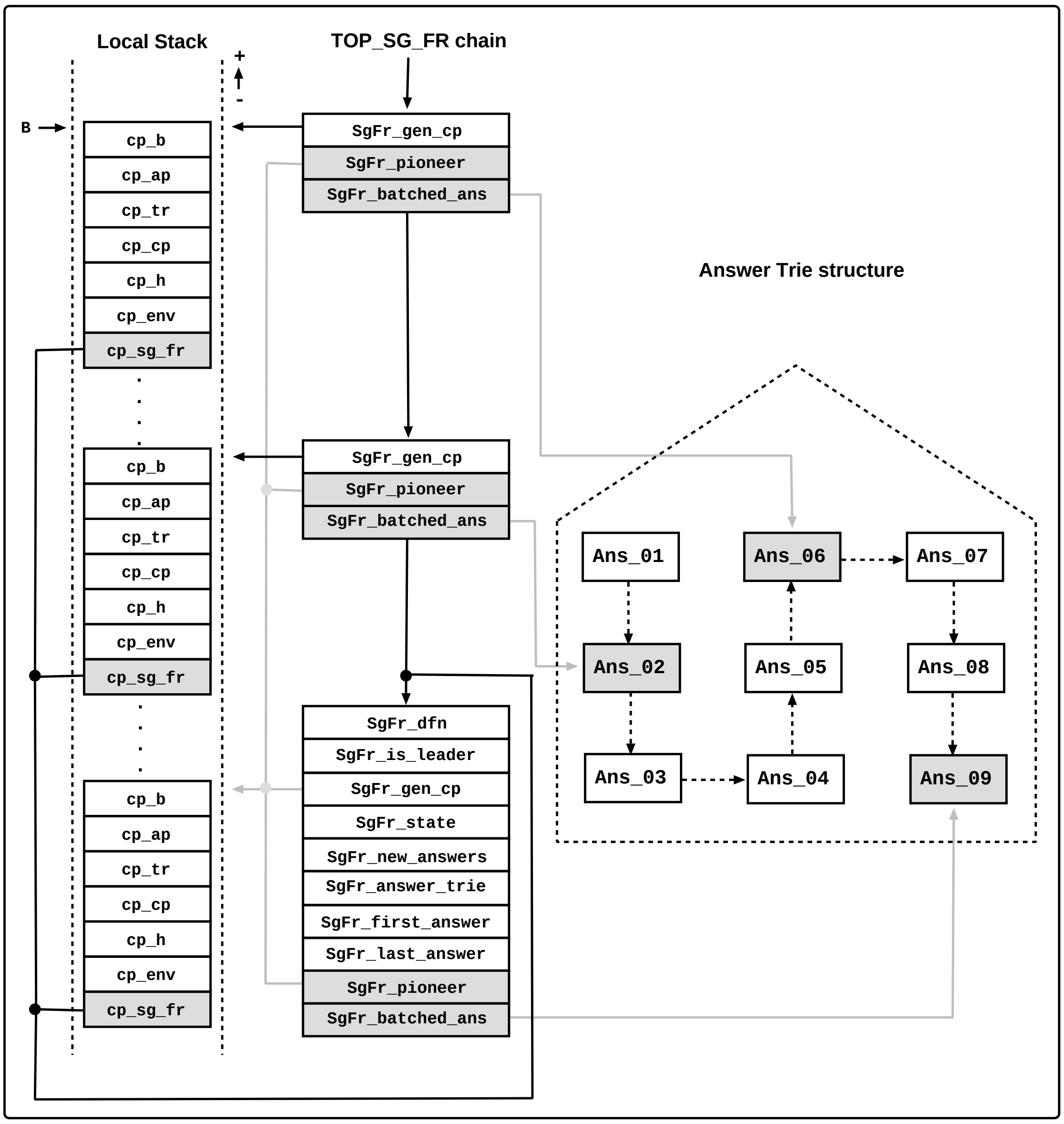


Figure 5.5: The DRE optimization with support for batched scheduling

most choice point.

The support of DRE optimization is then fully integrated with batched scheduling, without requiring major changes to the key tabling operations already implemented. It just involved changing the structure and the initialization of the subgoal frames to support the `SgFr_batched_ans` field on follower nodes, and changing the fix-point check operation to get the batched answer through the `TOP_SG_FR` chain, instead of the pioneer's subgoal frame.

5.4 Chapter Summary

This chapter described the most important ideas and difficulties for implementing the batched scheduling strategy on our linear tabling system. It started by presenting the high-level approach and by describing the batching of answers scheme. Then, it discussed the changes made to the major operations of our linear system and ended by describing the changes required to support the DRE optimization.

Chapter 6

Performance Analysis

In this chapter, we analyze the advantages and weaknesses of each linear tabling optimization, when used solely or combined with the others, and make a comparison between the suspension-based and linear-based mechanisms of the YapTab system. The environment for our experiments was an Intel(R) Core(TM)2 Quad CPU Q9550 2.83GHz with 4 GBytes of main memory and running the Linux kernel 2.6.32-24-PAE with Yap 6.0.7. For the calculation of the running times that we present next, each benchmark was executed three times and the results presented on this chapter are the average of those three executions.

6.1 Benchmarks

We will use three different sets of benchmarks for performance analysis. For the first two sets we will present an exhaustive discussion about the results obtained and for the third set we will just include the results as an appendix without further analysis.

The first set of benchmarks is a set of six different versions of the well-known `path/2` predicate, that computes the transitive closure in a graph, combined with several different configurations of `edge/2` facts. The six versions of the path predicate include two double recursive, two right recursive and two Left recursive definitions as presented on Fig. 6.1. Each pair has one definition with the recursive clause first and another with the recursive clause last.

Regarding the edge facts, we used three configurations: a pyramid, a cycle and a grid configuration (Fig. 6.2 shows an example for each configuration). We experimented

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

% Double Last
path(X,Z) :- edge(X,Z).
path(X,Z) :- path(X,Y), path(Y,Z).

% Right First
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Z) :- edge(X,Z).

% Right Last
path(X,Z) :- edge(X,Z).
path(X,Z) :- edge(X,Y), path(Y,Z).

% Left First
path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

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

Figure 6.1: The six versions of the *path/2* predicate

the pyramid and cycle configurations with depths 500, 1000 and 1500 and the grid configuration with depths 20, 30 and 40. We also experimented the Left recursive definition of the `path/2` predicate[1] with three different transition relation graphs usually used in Model Checking applications: the *i-protocol* (Iproto), *leader election* (Leader) and *sieve* specifications.

The second set of benchmarks is a small variant of the path problem, suggested by David S. Warren as a way to stress the evaluation of a linear tabling system, that leads to successive re-evaluations of the same SCC with the solutions being found only on

[1]We didn't show results for the Right and Double recursive definitions of the *path/2* predicate because they took more than 5 hours to execute in YapTab and thus we aborted their execution.

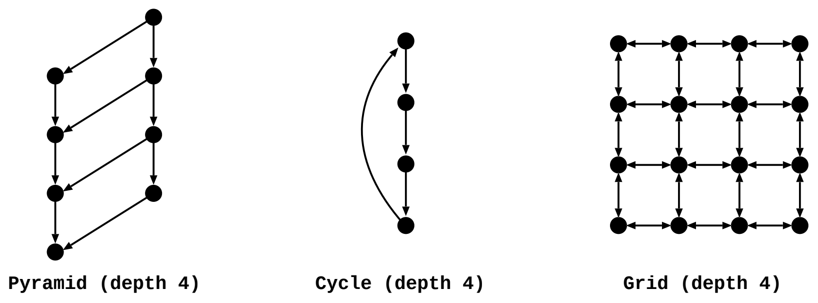


Figure 6.2: Edge configurations for path definitions

leaf nodes. Figure 6.3 presents an example of the Prolog code for these benchmarks, that we named Warren tests, with the transition graph, defined by the predicate `edge/3`, with depth 6. The transition graph is defined by the function:

$$edge/3 = \begin{cases} edge(2k, a, 2k+1) & for \;\; 0 <= k <= (depth/2) - 1 \\ edge(2k+1, b, 2k+2) & for \;\; 0 <= k <= (depth/2) - 1 \end{cases}$$

and we have used the depths 3000, 6000, 9000 and 12000 on the experimental tests.

```
:- table path/2.
path(X,Z) :- path(X,Y), edge(Y,a,Z).
path(X,Z) :- path(X,Y), edge(Y,b,Z).
path(X,X).

% Edge depth is 6
edge(0,a,1).
edge(1,b,2).
edge(2,a,3).
edge(3,b,4).
edge(4,a,5).
edge(5,b,6).
```

Figure 6.3: An example of the Warren tests with depth 6

The third and last set of benchmarks were obtained from the OpenRuleBench community. We have submitted our system to all their tests [16], but on this thesis we have only included a small part of them that are detailed on Appendix B.

6.2 Local Scheduling Results

We begin the description of the experimental results for local scheduling with the path problem. Table 6.1 shows the running time average ratios for the comparison of standard linear tabling against the several optimizations. The values on the table have the following meaning: a 1.00 value means that standard linear tabling has the same running time of the corresponding optimization, a value higher than 1.00 means that the optimization is better and lower means that it is worst than standard linear tabling.

The table is divided into eight columns. The first column identifies the program used. In bold text, it refers how the *path/2* predicate is recursively defined, and for each definition of the *path/2* predicate follows the edge configurations Cycle, Grid and Pyramid. In order to present an higher picture of the results, the results presented for these configurations are the average results of the different depths for each edge type. For example, the results presented for the Grid configuration are the average of the depths 20, 30 and 40. On Appendix A.1 the reader can find all the detailed results. The two Left recursive definitions of *path/2* have three extra edge definitions, which correspond to the Model Checking configurations mentioned before. The remaining columns match the corresponding optimizations. The DRA, DRE and DRS columns correspond to solely optimizations and the DRA+DRE, DRA+DRS, DRE+DRS and *All* (DRA+DRE+DRS) correspond to the combined optimizations. On the last line of the table, it is described the average of the results per column.

Analyzing the general picture of this first set of results, we can observe that the *All* (DRA+DRE+DRS) optimization has the best results with an average result of 1.26, which represents that, on average it is about 26% faster than the standard evaluation. The second best optimization was the DRA+DRS with 1.24. In general, all the optimizations have positive results, but in fact some of them have more consistent results than others. An example of this situation is the DRA and DRE optimizations. The DRA presents more diverse results with values between 1.09 and 1.71, and good results on the Double and Right definitions of the *path/2* predicate. The DRE presents more consistent results with values between 1.07 and 1.28, and has a lower performance on the right definition of *path/2*.

In order to better understand these results, i.e., how they are affected by the different optimizations and, if in fact, the optimizations are being used during the evaluation, we have collected some internal statistics of the evaluation. On those statistics, we

Table 6.1: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Double First							
Cycle	1.16	1.15	1.15	1.16	1.14	1.15	1.17
Grid	1.11	1.11	1.10	1.10	1.09	1.09	1.09
Pyramid	1.02	1.02	1.00	1.02	1.02	1.01	1.02
Double Last							
Cycle	1.12	1.12	1.13	1.12	1.11	1.13	1.14
Grid	1.17	1.18	1.16	1.16	1.16	1.17	1.17
Pyramid	1.13	1.13	1.14	1.14	1.12	1.13	1.14
Right First							
Cycle	1.18	1.02	1.28	1.22	1.64	1.27	1.56
Grid	1.12	1.06	1.27	1.15	1.42	1.30	1.49
Pyramid	1.71	1.08	1.10	1.72	1.72	1.08	1.74
Right Last							
Cycle	1.37	1.16	1.43	1.43	1.76	1.47	1.57
Grid	1.14	1.03	1.27	1.14	1.43	1.28	1.46
Pyramid	1.53	1.07	1.07	1.58	1.62	1.03	1.56
Left First							
Cycle	1.12	1.15	1.14	1.14	1.14	1.14	1.16
Grid	1.30	1.28	1.27	1.31	1.28	1.33	1.35
Pyramid	1.12	1.19	1.17	1.18	1.14	1.17	1.18
Iproto	1.09	1.08	1.09	1.10	1.10	1.15	1.13
Leader	1.09	1.09	1.06	1.04	1.05	1.13	1.10
Sieve	1.09	1.07	1.06	0.99	1.01	1.10	1.05
Left Last							
Cycle	1.15	1.15	1.16	1.22	1.15	1.17	1.18
Grid	1.28	1.27	1.24	1.29	1.28	1.35	1.35
Pyramid	1.09	1.15	1.11	1.15	1.09	1.14	1.16
Iproto	1.13	1.13	1.13	1.15	1.14	1.19	1.18
Leader	1.12	1.12	1.09	1.08	1.07	1.16	1.13
Sieve	1.08	1.07	1.07	0.99	1.01	1.09	1.06
Average	1.18	1.12	1.15	1.19	1.24	1.18	1.26

are interested on particular points of the evaluation where the optimizations may take effect. In particular, we are interested on the number of tabled nodes allocated per evaluation, on the number of answers consumed by generator nodes, the number of alternatives evaluated and on the number of SCC evaluations. In Table 6.2, we show statistical information for the particular evaluation of the path problem using the Grid configuration with depth 40 in order to compare the standard evaluation against the optimizations.

Again, the table has eight columns. The first column is the subject of the statistic. The "*Tabled Nodes*" item represents the number of tabled nodes allocated per evaluation. In this item, we count all the generator and consumer nodes and, whenever the DRE optimization is enabled, we also count the follower nodes. The "*Answers*" item represents the number of answers that are consumed by generator nodes corresponding to non-leader subgoals. The "*Alternatives*" item represents the number of alternatives explored during the evaluation. Recall that the objective of the DRA optimization is to reduce this value to a minimum. The last item is the "*SCC Eval*" and it counts the number of evaluation rounds of all SCCs.

The remaining columns show the values gathered for each optimization. These values represent again the comparison with standard evaluation, but now instead of analyzing ratios as for the previous table, we are interested in analyzing the total values themselves. Thus, a zero value represents evaluations where the standard evaluation is equal to the optimization at hand, negative numbers represent evaluations where the standard evaluation is worst than the optimization and positive numbers represent evaluations where standard evaluation is better that the optimization. For example, in Table 6.2 for the Right First definition of the *path/2* predicate, we can observe that whenever the DRE optimization is present, the evaluation allocates less 3,121 nodes than the standard evaluation.

We will now turn our attention to the confrontation of the data shown on Table 6.1 with the data shown on Table 6.2, and, for that, we will focus on the right first definition of the *path/2* predicate. We will leave the considerations for the remaining definitions of the *path/2* predicate for the reader as the same analysis can be applied.

We begin the analysis by the solely used optimizations. The DRA optimization has a running time ratio of 1.12, and as we can see on Table 6.2 it executes 33,601 less alternatives and one less time the SCC. The DRE optimization, has a running time ratio of 1.06, and it allocates less nodes, consumes less answers on non-leader generator nodes, executes less alternatives (even thought the DRA and DRS are not in use) and

Table 6.2: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the path problem for the Grid configuration with depth 40

Programs	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Double First							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-5,120,000	0	-5,120,000	-5,120,000	-5,120,000
Alternatives	-3,200	-3,200	0	-3,201	-3,200	-3,200	-3,201
SCC Eval	0	-1,601	0	-1,601	0	-1,601	-1,601
Double Last							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-5,120,000	0	-5,120,000	-5,120,000	-5,120,000
Alternatives	-1,601	0	0	-1,601	-1,601	0	-1,601
SCC Eval	0	0	0	0	0	0	0
Right First							
Tabled Nodes	0	-3,121	0	-3,121	0	-3,121	-3,121
Answers	0	-1,934,915	-47,456,815	-1,934,915	-47,456,815	-47,456,815	-47,456,815
Alternatives	-33,601	-3,199	0	-4,001	-33,601	-3,199	-4,001
SCC Eval	-1	-1	0	-2	-1	-1	-2
Right Last							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-45,521,900	0	-45,521,900	-45,521,900	-45,521,900
Alternatives	-32,002	0	0	-32,002	-32,002	0	-32,002
SCC Eval	-1	0	0	-1	-1	0	-1
Left First							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-2,560,000	0	-2,560,000	-2,560,000	-2,560,000
Alternatives	-1	-1	0	-2	-1	-1	-2
SCC Eval	0	-1	0	-1	0	-1	-1
Left Last							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-2,560,000	0	-2,560,000	-2,560,000	-2,560,000
Alternatives	-1	0	0	-1	-1	0	-1
SCC Eval	0	0	0	0	0	0	0

evaluates one less time the SCC. The DRS optimization as a running time ratio of 1.27, which is the highest so far, and it consumes far less answers than the standard evaluation.

On the combined optimizations, the DRA+DRE optimization has a running time ratio of 1.15, which corresponds almost to the sum of the DRA and DRE solely optimizations. Using this optimization, the evaluation uses less nodes, answers and alternatives, and executes less times the SCC. But the number of the alternatives explored is far more higher than the DRA used solely. This can explain why the sum of both optimizations is 1.15 and not exactly 1.18.

The DRA+DRS optimization has a ratio of 1.42, and it seems that the benefits of both optimizations are fully present. In particular, it shows the same number of answers and alternatives that the respective DRS and DRA optimizations, show when used solely. The 1.42 ratio actually exceeds the sum of both optimizations used solely, which is 1.39. This could be explained by other unmeasured factors, such as the Prolog system spending less time on stack overflows recoveries.

The DRE+DRS optimization has a ratio of 1.30, but the sum of both optimizations used solely is 1.33. This is a very small difference, but a closer look to the consumed answers explains this number. In fact, the non consumed answers of the DRE optimization used solely, are included on the non consumed answers of the DRS optimization. So, for this particular evaluation, both optimizations are not fully orthogonal.

The *All* (DRA+DRE+DRS) optimization has a ratio of 1.49, which is the best of all combinations, with a positive difference from the sum of all optimizations used solely, which is 1.45. Even though, the number of alternatives explored is higher than the DRA optimization used solely, but the gain obtained on the other items seems to be sufficient to boost the overall gain of this optimization.

The second set of results are concerned with the evaluation of the tests proposed by David S. Warren. Table 6.3 shows the running time average ratios for the comparison of standard linear tabling against the several optimizations using depths 3000, 6000, 9000 and 12000.

Table 6.3: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the Warren tests (values higher than 1.00 mean that the optimization is better)

Programs / **Depth**	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
3000	1.35	0.48	1.23	0.60	1.22	0.54	0.55
6000	1.08	0.37	0.99	0.42	0.98	0.41	0.43
9000	1.11	0.39	1.02	0.46	1.00	0.43	0.45
12000	1.04	0.36	0.94	0.43	0.89	0.39	0.41
Average	1.15	0.40	1.05	0.48	1.02	0.44	0.46

Analyzing the general picture of the results, we can observe that whenever the DRE optimization is present (used solely or combined), the ratios are lower than 1.00. This means that, for this set of tests, this optimization clearly degrades the performance of the system. The optimizations DRA and DRS used solely have an average ratio of 1.15 and 1.05, respectively. The combination of both has an average ratio of 1.02, which is lower than each optimizations used solely.

Again, in order to help us to understand these results, Table 6.4 shows internal statistics of the evaluation of the Warren tests, comparing standard linear tabling with the optimizations.

Table 6.4: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the Warren tests

Programs / **Depth**	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
3000							
Tabled Nodes	0	3,000	0	3,000	0	3,000	3,000
Answers	0	0	-3,001	0	-3,001	-3,001	-3,001
Alternatives	-1,500	4,498	0	1,498	-1,500	4,498	1,498
SCC Eval	0	-1	0	-1	0	-1	-1
6000							
Tabled Nodes	0	6,000	0	6,000	0	6,000	6,000
Answers	0	0	-6,001	0	-6,001	-6,001	-6,001
Alternatives	-3,000	8,998	0	2,998	-3,000	8,998	2,998
SCC Eval	0	-1	0	-1	0	-1	-1
9000							
Tabled Nodes	0	9,000	0	9,000	0	9,000	9,000
Answers	0	0	-9,001	0	-9,001	-9,001	-9,001
Alternatives	-4,500	13,498	0	4,498	-4,500	13,498	4,498
SCC Eval	0	-1	0	-1	0	-1	-1
12000							
Tabled Nodes	0	12,000	0	12,000	0	12,000	12,000
Answers	0	0	-12,001	0	-12,001	-12,001	-12,001
Alternatives	-6,000	17,998	0	5,998	-6,000	17,998	5998
SCC Eval	0	-1	0	-1	0	-1	-1

The table explains almost all the reasons for the running times ratios presented in

Table 6.3, but also leaves one unanswered question, as we will observe later.

Lets analyze the results starting by the optimizations used solely. The DRA optimization executes less alternatives than the standard evaluation, and the gain on each test is half of the test's depth. This is the reason why the test have a general good performance.

Regarding the DRE optimization, the statistics explain why it has a very poor performance on these tests. The number of extra nodes allocated by the DRE optimization in comparison with standard evaluation is equal to the depth of the test. In addition, this optimization also executes much more alternatives than the standard evaluation. As the depth of the tests increases in 3000 units, the number of explored alternatives increases in 4500 units. This explains why the running time ratios for the depth 3000 is 0.48 and for depth 12000 is 0.36.

The DRS optimization, presents an average running time ratio of 1.05, which is a value closer to 1.00. The statistics show that using this optimization, the number of non consumed answers is proportional to the test's depth, yet this fact is not sufficient to ensure a good performance for the optimization.

Analyzing now the combined optimizations, we can observe whenever the DRE optimization is present (DRA+DRE, DRE+DRS and DRA+DRE+DRS), the number of nodes allocated and alternatives explored, is still higher than the standard evaluation. This explains why the running time average ratios are lower than 1.00. Regarding the DRA+DRS optimization, the statistics leave an unanswered question, because it shows that, using this optimization, the evaluation consumes less alternatives and answers, but in fact the running time ratios do not show this advantage.

6.3 Batched Scheduling Results

On this section, we present the results for the same set of programs but using the batched scheduling strategy. Since the DRS optimization was not implemented with this strategy, no results for it were included on the following tables. The tables with statistics do not include also the *Answers* item, which is mostly regarded with consumption of answers when using the DRS optimization.

Table 6.5 shows the running time average ratios comparing standard linear tabling against the DRA, DRE and DRA+DRE optimizations for the path problem. Again, as for Table 6.1, the results presented here are the average results of the different depths for each configuration. On Appendix A.2 the reader can find the full details about the results.

The average results are 1.19, 1.03 and 1.23 for the optimizations DRA, DRE and DRA+DRE, respectively. The DRA optimization used solely presents borderline results on the Double First and Double Last definitions of the *path/2* predicate, with values close to 1.00. The results obtained for the Left First and Left Last are a little better, but even so, the gain is around 10%. On the Right First and Right Last definitions, the results are very good. The gain achieved for the Cycle and Grid configurations was between 30% and 40% and for the Pyramid configurations was between 80% and 105%. We can observe also that the best results on all the definitions of the path predicate were achieved on the Pyramid configurations (on the Left definitions, the best result is shared with the Iproto configuration).

The DRE optimization used solely presents very good results for the Double First definition of the *path/2* predicate. On the Cycle configuration it achieves a 2.04 ratio (represents 104% of gain), on the Grid configuration it achieves a 1.80 ratio (represents 80% of gain) and on the Pyramid configuration it achieves a 2.02 ratio (represents 102% of gain). On the remaining definitions of the *path/2* predicate, the DRE optimization present borderline results on the Double Last definition and poor results on the other definitions.

The DRA+DRE optimization suffers from the side effects of using the DRE optimization. For the Double First definition of the *path/2* predicate it presents values of 2.05, 1.81 and 2.07, for the Cycle, Grid and Pyramid configurations, respectively. For the Double Last definition the results shows the same pattern as for the Double First, but with less gains over the standard evaluation. On the other hand, the results for the remaining definitions, show poor and always worst results than the results obtained with the DRA optimization used solely.

Again, in order to help us to understand these running time average results for the path problem, we next present on Table 6.6 statistical data for the evaluation of the path predicate using the Grid configuration, with depth 40.

If we focus our attention on the Double First definition, the results on the table show that the DRA optimization used solely executes less alternatives and evaluates less times the SCC during the evaluation, when compared with the standard evaluation. The DRE optimization used solely allocates less 2,554,043 nodes than the standard evaluation, consumes less alternatives than the standard evaluation, but more than

Table 6.5: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRA + DRE**
Double First			
Cycle	1.02	2.04	2.05
Grid	1.01	1.80	1.81
Pyramid	1.01	2.02	2.07
Double Last			
Cycle	1.02	1.03	1.04
Grid	1.02	1.03	1.08
Pyramid	1.07	1.07	1.08
Right First			
Cycle	1.30	0.95	1.23
Grid	1.40	1.00	1.30
Pyramid	2.05	0.87	1.53
Right Last			
Cycle	1.34	0.94	1.21
Grid	1.35	1.02	1.32
Pyramid	1.80	0.89	1.44
Left First			
Cycle	1.13	0.78	1.02
Grid	1.07	0.72	1.06
Pyramid	1.17	0.83	1.11
Iproto	1.17	0.65	1.15
Leader	1.10	0.46	1.06
Sieve	1.12	0.59	1.02
Left Last			
Cycle	1.05	0.95	0.98
Grid	1.01	0.98	0.98
Pyramid	1.09	1.04	1.06
Iproto	1.09	1.05	1.07
Leader	1.02	1.02	0.97
Sieve	1.08	1.07	0.99
Average	1.19	1.03	1.23

Table 6.6: Statistics for batched scheduling comparing standard linear tabling against the several optimizations using the path problem for the Grid configuration with depth 40

Programs	**DRA**	**DRE**	**DRA + DRE**
Double First			
Tabled Nodes	0	-2,554,043	-2,554,043
Alternatives	-3,202	-1,600	-1,601
SCC Eval	-2	-1,602	-1,602
Double Last			
Tabled Nodes	0	0	0
Alternatives	-1,601	0	-1,601
SCC Eval	0	0	0
Right First			
Tabled Nodes	0	803	803
Alternatives	-3,201	0	-81
SCC Eval	-2	-1	-2
Right Last			
Tabled Nodes	0	0	0
Alternatives	-3,201	0	-3,201
SCC Eval	-1	0	-1
Left First			
Tabled Nodes	0	1	1
Alternatives	-2	0	-2
SCC Eval	-1	-1	-1
Left Last			
Tabled Nodes	0	0	0
Alternatives	-1	0	-1
SCC Eval	0	0	0

the DRA optimization used solely, and the number of evaluation rounds of the SCC is reduced in 1,602 rounds. The results presented by the DRA+DRE optimization are identical to the DRE optimization used solely. It allocates the same number of nodes, evaluates the same number of rounds the SCC and the difference is that it executes one less alternative.

Table 6.7 shows the running time average ratios for the comparison of the Warren tests using batched scheduling. As for local scheduling, we will use the depths 3000, 6000, 9000 and 12000.

Table 6.7: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the Warren tests (values higher than 1.00 mean that the optimization is better)

Programs / **Depth**	**DRA**	**DRE**	**DRA + DRE**
3000	1.01	29.00	174.00
6000	1.01	50.57	354.00
9000	1.00	62.31	741.50
12000	1.01	74.15	810.00
Average	1.01	54.01	519.88

The average ratios of the running times are 1.01, 54.01 and 519.88 for the DRA and DRE optimizations used solely, and the combined DRA+DRE optimization, respectively. The DRA optimization used solely presents borderline results on all depths.

The DRE optimization shows excellent results. For depth 3000 it has a ratio of 29.00, for the depth 6000 a ratio of 50.57, for the depth 9000 a ratio of 62.31 and for the depth 12000 a ratio of 74.15. The ratio difference between depths 3000 and 6000 is around 20.00, and the difference between depths 6000 and 9000 and between the depths 9000 and 12000, is around 12.00. This indicates that for deeper tests we would expect an increase the ratio in a factor of around 4.00 per 1000 depth edges.

The combination of DRA and DRE optimizations, boosts the gain on the running time ratios. The running time ratios are 174.00, 354.00, 741.50 and 810.00, respectively for depths 3000, 6000, 9000 and 12000. The difference between ratios is not as consistent as for the DRE optimization used solely, but for deeper tests we would expect an increase in a factor of about 35.00 per 1000 depth edges.

Table 6.8 shows the statistics for the evaluation of the Warren tests. The results on Table 6.8 help us to understand why the DRE optimization, is so effective for this particular set of tests. The DRE optimization used solely allocates less nodes, executes less alternatives and evaluates less times the SCC. The DRA+DRE combination slightly decreases the number of allocated nodes, executed alternatives and SCC evaluations. On both optimizations, as the test's depth increases, the differences to the standard evaluation also increases on all items. The number of allocated nodes and executed alternatives decreases in a factor of 1 per depth edge and the number of evaluations of the SCC decreases in a factor of 2 rounds per 3 depth edges.

Table 6.8: Statistics for batched scheduling comparing standard linear tabling against the several optimizations using the Warren tests

Programs / **Depth**	**DRA**	**DRE**	**DRA + DRE**
3000			
Tabled Nodes	0	-2,920	-2,996
Alternatives	-1,501	-2,922	-3,000
SCC Eval	-1,500	-4,461	-4,499
6000			
Tabled Nodes	0	-5,888	-5,996
Alternatives	-3,001	-5,890	-6,000
SCC Eval	-3,000	-8,945	-8,999
9000			
Tabled Nodes	0	-8,862	-8,996
Alternatives	-4,501	-8,864	-9,000
SCC Eval	-4,500	-13,432	-13,499
12000			
Tabled Nodes	0	-11,842	-11,996
Alternatives	-6,001	-11,844	-12,000
SCC Eval	-6,000	-17,922	-17,999

6.4 Comparison with YapTab

On this section, we compare our linear tabling system with YapTab's suspension-based mechanism. With this comparison, we want to analyze the advantages and weaknesses of our linear tabling system when compared with a more sophisticated system. For this purpose, we used the six definitions of the *path/2* predicate and the Warren tests. The reader can find also the comparison for the OpenRuleBench tests on Appendix B. As in all previous tables, we will use again the standard linear running times as base results for our ratios.

We begin with Table 6.9, showing the running time average ratios for the comparison of standard linear tabling with YapTab's suspension-based mechanism and the best linear optimization for the path problem. The table has five columns and it is divided in two main blocks, one for local scheduling and the other for batched scheduling. The first column is the definition of the program. The second and third columns show the results for local scheduling and the fourth and fifth columns show the results for batched scheduling. The second and forth columns show the ratio for the results obtained with YapTab, and the third and fifth columns, show the ratio for the results obtained by the best linear optimization when compared with standard linear tabling.

For local scheduling, the results show that YapTab is always faster than standard linear tabling or its optimizations. For the Double First and Double Last definitions, the highest difference is for the Pyramid configurations. In these configurations, YapTab has ratios of 2.04 and 2.98, and the DRA optimization used solely, which is the best optimization for linear tabling, has ratios of 1.02 and 1.14, respectively. For the Right First and Right Last definitions, the biggest difference is on the Grid configurations. YapTab has results rounding the 5.20 times faster than standard linear tabling, while the best linear optimization DRA+DRE+DRS has results rounding 1.47. For the Left First and Left Last definitions, the biggest difference is again on the Grid configurations. On the Iproto, Leader and Sieve configurations, YapTab has ratios rounding 2.30 and the best linear optimization, which is DRE+DRS, has ratios rounding 1.10. Generally speaking, for the local scheduling strategy, the gain achieved using the optimizations, is not enough to reach the performance of YapTab's suspension-based mechanism. If we compare the success of the optimizations, by the number of times it appears as the best optimization, then we can conclude that the best is All (DRA+DRE+DRS) optimization.

On the other hand, for the batched scheduling, we can observe that for the Double First definition of the *path/2* predicate, the linear optimization DRA+DRE presents similar ratios to YapTab. For the remaining tests, YapTab is about 2.05 times faster than standard linear tabling, while the best linear optimization is, on average, around 1.30 times faster. Analyzing the performance of the linear optimizations, the most successful optimization was the DRA used solely for both Left and Right definitions of the *path/2* and for the Double definitions was the DRA+DRE.

Concerning the second set of tests, Table 6.10 shows the running time average ratios for the comparison of standard linear tabling with YapTab's suspension-based mechanism, and the best linear optimization using local and batched scheduling strategies.

Table 6.9: Running time ratios for local and batched scheduling comparing standard linear tabling against YapTab and the best linear optimization using the path problem

Programs	**Local Scheduling**		**Batched Scheduling**	
	YapTab	**Best Linear (Opt)**	**YapTab**	**Best Linear (Opt)**
Double First				
Cycle	1.96	1.17 (All)	2.04	2.05 (DRA+DRE)
Grid	1.85	1.11 (DRA)	2.07	1.81 (DRA+DRE)
Pyramid	2.04	1.02 (DRA)	2.02	2.07 (DRA+DRE)
Double Last				
Cycle	1.92	1.14 (All)	2.04	1.04 (DRA+DRE)
Grid	1.93	1.18 (DRE)	2.07	1.08 (DRA+DRE)
Pyramid	2.98	1.14 (All)	2.16	1.08 (DRA+DRE)
Right First				
Cycle	1.43	1.64 (DRA+DRS)	2.09	1.30 (DRA)
Grid	5.25	1.49 (All)	2.10	1.40 (DRA)
Pyramid	1.78	1.74 (All)	1.99	2.05 (DRA)
Right Last				
Cycle	1.77	1.76 (DRA+DRS)	1.82	1.34 (DRA)
Grid	5.14	1.46 (All)	2.25	1.35 (DRA)
Pyramid	1.59	1.62 (DRA+DRS)	1.99	1.80 (DRA)
Left First				
Cycle	1.92	1.16 (All)	2.09	1.13 (DRA)
Grid	2.80	1.35 (All)	2.14	1.07 (DRA)
Pyramid	1.99	1.19 (DRE)	2.26	1.17 (DRA)
Iproto	2.23	1.15 (DRE+DRS)	2.57	1.17 (DRA)
Leader	2.28	1.13 (DRE+DRS)	2.31	1.10 (DRA)
Sieve	2.26	1.10 (DRE+DRS)	2.34	1.12 (DRA)
Left Last				
Cycle	1.99	1.22 (DRA+DRE)	1.94	1.05 (DRA)
Grid	2.48	1.35 (All)	1.89	1.01 (DRA)
Pyramid	1.96	1.16 (All)	2.23	1.09 (DRA)
Iproto	2.30	1.19 (DRE+DRS)	2.33	1.09 (DRA)
Leader	2.34	1.16 (DRE+DRS)	2.13	1.02 (DRA)
Sieve	2.26	1.09 (DRE+DRS)	2.26	1.08 (DRA)

Table 6.10: Running time ratios for local and batched scheduling comparing standard linear tabling against YapTab and the best linear optimization using the Warren tests

Programs	**Local Scheduling**		**Batched Scheduling**	
Depths	**YapTab**	**Best Linear (Opt)**	**YapTab**	**Best Linear (Opt)**
3000	384.00	1.35 (DRA)	348.00	174.00 (DRA+DRE)
6000	1,264.00	1.08 (DRA)	354.00	354.00 (DRA+DRE)
9000	2,992.00	1.11 (DRA)	810.00	741.50 (DRA+DRE)
12000	1,288.00	1.04 (DRA)	1,483.00	810.00 (DRA+DRE)

For the local scheduling strategy, the results show that the best linear optimization was the DRA used solely however, these best results have a huge difference from YapTab's suspension-based results. All the tests with depths equal or higher than 6000, have ratios higher that 1,000.00 with YapTab's.

For the batched scheduling strategy, the results show again an huge difference between YapTab's running time results and standard linear tabling. But for the best linear optimization, which is DRA+DRE in all cases, the difference is not so huge. In fact, some of them have similar performance to YapTab. For both YapTab and the best linear optimization, the gain tends to increase as the depth configuration increases despite the fact that YapTab's gain ratio is higher than the DRA+DRE linear optimization. Thus, for deeper configurations, it would be expectable that YapTab would increase the difference to the DRA+DRE optimization.

6.5 Chapter Summary

In this chapter, we analyzed and compared the performance of the standard linear mechanism with the several optimizations, for the local and batched scheduling strategies, and the performance between the best linear-based mechanism with the suspension-based mechanism of YapTab.

Chapter 7

Conclusions and Future Work

In this chapter, we conclude this thesis by summarizing the work developed on the design, implementation and performance evaluation of the linear tabling mechanisms created for the YapTab system.

7.1 Conclusions

This thesis had two main goals. The first goal was to implement on YapTab an efficient linear tabling mechanism which, in theory, could compete with suspension-based mechanisms for both local and batched scheduling mechanisms. The second goal was that our system should be as robust as possible, meaning that it should be capable of correctly evaluate an huge class of problems written in Prolog.

To attend both goals we started the work by creating a test suite engine. Actually, the engine has about 5 GBytes of information between several different tests and their solutions/tables produced. The engine is capable of comparing running time results, and test the correctness of the program's solutions and tables obtained for the YapTab, XSB and B-Prolog systems. The programs on the test suite include the path problem with different definitions of the *path/2* predicate and different transition graphs, the problem proposed by David H. Warren with different edges, model checking tests, basic tests to evaluate particular situations, and the tests mentioned on chapter 6 obtained from the OpenRuleBench community.

The second goal of our work, which was the correctness of the system, is thus actually supported by the results produced by the test suite. Actually, for the local and batched strategies, there was only one test, the recursion wine test obtained from the Open Rule Bench community, that fails with on our linear tabling system. All the remaining tests had successful results.

For the first goal, we have presented a new optimization for linear tabled evaluation of logic programs using the local scheduling strategy, named DRS, and a framework, on top of the Yap system, that integrates and supports the combination of the DRS strategy with two other linear tabling optimizations, the DRE and DRA optimizations. For the batched scheduling strategy, our framework includes the support for DRA and DRE optimizations. We discussed how these strategies can optimize different aspects of a tabled evaluation and we presented some implementation details of their integration, with particular focus on the table space data structures and on the tabling operations.

The performance of our linear tabling system highly depends on using the correct combination of optimizations for the problem at hand. As observed on the previous chapter, different problems might have different results the same optimization (or combination of optimizations), and the responsibility of choosing the best optimization is given to the programmer. The performance of each optimization can be summarized as follows:

- In general, the DRA optimization had good results. It reduces the running times for programs with loop clauses, and if these type of clauses are not present, it does not add any extra overhead to the evaluation.
- The DRE optimization can have very good or very bad results. It depends on the type of the problem which is being evaluated. For example, for the Warren tests, it can be considered a very good optimization when used with batched scheduling, but can also a very bad optimization when used with local scheduling.
- The DRS optimization had also good results. It showed that the strategy of avoiding the consumption of non-looping solutions in re-evaluation rounds can be quite effective for programs that can benefit from it, with insignificant costs for the other programs.
- The combined optimization DRA+DRS also obtained good results. It showed that both optimizations can be combined without jeopardizing the performance of each other.

- The combined optimizations with DRE enabled can have good or bad results, because they are too dependent on the performance of the DRE optimization for the particular test being considered.

Regarding the performance comparison between our linear-based tabling system and YapTab's suspension-based engine, the results obtained with our approach are very interesting and very promising. Our experiments confirmed the idea that, in general, suspension-based mechanisms obtain better results than linear tabling and that the difference between both mechanisms depends of the specifics of the problem to be evaluated. However, the commonly referred weakness of linear tabling of doing a huge number of redundant computations for computing fix-points was not such a problem in most of our experiments.

We thus argue that linear-based tabling mechanisms have two major advantages when compared with suspended-based tabling. The first is that it is easier to implement and thus it can be a good and first alternative to incorporate tabling into a Prolog system without tabling support. The second is that by using the correct linear tabling optimization, the difference between both approaches can be highly reduced. Moreover, as linear tabling mechanisms use less memory space, this can have positive effects on intensive memory usage problems.

7.2 Future Work

We next suggest some topics for future work:

More experimentation. Explore the impact of applying our strategies to more complex problems, seeking real-world experimental results allowing us to improve and consolidate even further our current implementation.

Support for negation. A wide range on problems that use tabling require the possibility to manipulate negative subgoals. Extending our implementation with this feature can be one major step forward to make it usable for a large community.

Support for multi-threading and parallelism. Since the evaluation of programs in our linear tabling engine is less complex than the evaluation using a suspension-based engine, it should be interesting to study how several linear tabled evaluations can run concurrently within such a model and take advantage of the different linear tabling optimizations. Also, it should be interesting to compare those results with the results already obtained with suspension-based mechanisms for multi-threading [33] and parallelism [26].

7.3 Final Remark

The research involved analyzing the execution models for linear tabling and, in particular, the ones already implemented on other tabling systems. But, in fact, the first implementation of a tabling engine on a Prolog system was a suspended-based tabling engine. In our opinion, the linear-based tabling engines became hostages of this fact and most of the actual research on tabling in done on suspended-based mechanisms. We thus argue that, there is still too much work that can be done for this type of mechanisms in order to increase their performance, and this thesis is only a small step in that direction.

Appendix A

Path Tests

A.1 Local Scheduling

Table A.1: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the double definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Double First							
Cycle							
500	1.26	1.26	1.25	1.26	1.25	1.26	1.27
1000	1.06	1.06	1.05	1.06	1.06	1.06	1.08
1500	1.15	1.15	1.15	1.15	1.12	1.15	1.17
Pyramid							
500	1.01	1.01	1.00	1.01	1.00	1.00	1.01
1000	1.02	1.02	1.02	1.02	1.01	1.01	1.02
1500	1.02	1.02	1.00	1.01	1.03	1.02	1.05
Grid							
20	1.24	1.24	1.22	1.24	1.22	1.24	1.19
30	1.03	1.04	1.03	1.03	1.03	1.00	1.04
40	1.04	1.04	1.03	1.03	1.03	1.04	1.03
Double Last							
Cycle							
500	1.17	1.17	1.17	1.17	1.13	1.18	1.19
1000	1.12	1.12	1.13	1.12	1.12	1.13	1.14
1500	1.08	1.07	1.08	1.07	1.07	1.08	1.10
Pyramid							
500	1.14	1.14	1.12	1.14	1.12	1.13	1.14
1000	1.15	1.15	1.15	1.15	1.14	1.14	1.13
1500	1.12	1.12	1.14	1.12	1.11	1.13	1.15
Grid							
20	1.41	1.41	1.40	1.40	1.40	1.39	1.42
30	1.08	1.09	1.08	1.05	1.07	1.08	1.06
40	1.03	1.04	1.01	1.03	1.02	1.03	1.04

Table A.2: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the right definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Right First							
Cycle							
500	1.15	1.03	1.28	1.21	1.74	1.28	1.56
1000	1.25	1.01	1.30	1.23	1.65	1.27	1.62
1500	1.14	1.03	1.26	1.22	1.53	1.27	1.52
Pyramid							
500	1.84	1.13	1.14	1.84	1.82	1.08	1.76
1000	1.66	1.06	1.08	1.68	1.71	1.10	1.82
1500	1.63	1.05	1.06	1.65	1.65	1.05	1.64
Grid							
20	1.14	1.07	1.21	1.18	1.39	1.29	1.49
30	1.12	1.05	1.28	1.14	1.42	1.31	1.49
40	1.09	1.05	1.31	1.12	1.44	1.30	1.49
Right Last							
Cycle							
500	1.43	1.26	1.58	1.60	1.89	1.64	1.84
1000	1.42	1.18	1.44	1.46	1.79	1.49	1.28
1500	1.26	1.04	1.28	1.24	1.60	1.30	1.60
Pyramid							
500	1.61	1.16	1.16	1.72	1.84	1.08	1.62
1000	1.50	1.01	1.01	1.55	1.54	1.00	1.58
1500	1.49	1.02	1.03	1.49	1.48	1.03	1.47
Grid							
20	1.16	1.01	1.19	1.18	1.37	1.21	1.44
30	1.12	1.02	1.26	1.11	1.43	1.27	1.44
40	1.14	1.06	1.34	1.14	1.48	1.35	1.50

Table A.3: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the left definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Left First							
Cycle							
500	1.07	1.11	1.11	1.11	1.17	1.07	1.07
1000	1.24	1.26	1.22	1.24	1.22	1.28	1.31
1500	1.05	1.07	1.09	1.07	1.04	1.07	1.11
Pyramid							
1000	1.12	1.17	1.17	1.18	1.13	1.17	1.13
1500	1.10	1.16	1.13	1.14	1.10	1.15	1.16
500	1.14	1.25	1.22	1.22	1.19	1.19	1.25
Grid							
20	1.75	1.70	1.70	1.75	1.70	1.75	1.81
30	1.11	1.08	1.07	1.10	1.09	1.15	1.13
40	1.05	1.07	1.05	1.07	1.06	1.10	1.12
Model Checking							
iproto	1.09	1.08	1.09	1.10	1.10	1.15	1.13
leader	1.09	1.09	1.06	1.04	1.05	1.13	1.10
sieve	1.09	1.07	1.06	0.99	1.01	1.10	1.05
Left Last							
Cycle							
500	1.11	1.07	1.17	1.23	1.11	1.11	1.17
1000	1.19	1.20	1.16	1.26	1.15	1.24	1.19
1500	1.14	1.18	1.14	1.18	1.17	1.16	1.18
Pyramid							
500	1.11	1.19	1.11	1.16	1.14	1.16	1.16
1000	1.11	1.15	1.15	1.18	1.11	1.17	1.18
1500	1.05	1.10	1.06	1.10	1.03	1.10	1.12
Grid							
20	1.73	1.68	1.63	1.73	1.73	1.84	1.84
30	1.10	1.11	1.09	1.13	1.10	1.15	1.12
40	1.02	1.02	1.00	1.01	1.00	1.07	1.08
Model Checking							
iproto	1.13	1.13	1.13	1.15	1.14	1.19	1.18
leader	1.12	1.12	1.09	1.08	1.07	1.16	1.13
sieve	1.08	1.07	1.07	0.99	1.01	1.09	1.06

A.2 Batched Scheduling

Table A.4: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the double definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRA + DRE**
Double First			
Cycle			
500	1.03	2.05	2.06
1000	1.01	2.03	2.05
1500	1.01	2.02	2.04
Pyramid			
500	1.01	1.99	2.03
1000	1.01	2.00	2.06
1500	1.01	2.08	2.10
Grid			
20	1.01	1.86	1.87
30	1.02	1.77	1.79
40	1.02	1.76	1.78
Double Last			
Cycle			
500	1.05	1.07	1.09
1000	1.01	1.01	1.02
1500	1.01	1.02	1.02
Pyramid			
500	1.14	1.14	1.15
1000	1.05	1.05	1.05
1500	1.02	1.02	1.02
Grid			
20	1.01	1.02	1.14
30	1.02	1.03	1.06
40	1.03	1.04	1.05

Table A.5: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the right definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRA + DRE**
Right First			
Cycle			
500	1.29	0.95	1.23
1000	1.27	0.93	1.25
1500	1.35	0.95	1.22
Pyramid			
500	2.03	0.87	1.51
1000	2.15	0.89	1.55
1500	1.98	0.85	1.53
Grid			
20	1.39	1.02	1.25
30	1.42	1.01	1.33
40	1.39	0.96	1.32
Right Last			
Cycle			
500	1.28	0.93	1.17
1000	1.39	0.96	1.25
1500	1.33	0.92	1.21
Pyramid			
500	1.45	0.89	1.28
1000	1.96	0.89	1.51
1500	2.00	0.88	1.53
Grid			
20	1.38	1.06	1.34
30	1.33	1.00	1.30
40	1.34	1.00	1.30

Table A.6: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the left definition of the path problem (values higher than 1.00 mean that the optimization is better)

Programs	**DRA**	**DRE**	**DRA + DRE**
Left First			
Cycle			
500	1.26	0.83	1.10
1000	1.02	0.74	0.96
1500	1.12	0.78	1.00
Pyramid			
500	1.08	0.78	0.97
1000	1.03	0.71	0.98
1500	1.40	1.02	1.37
Grid			
20	1.16	0.77	1.16
30	1.01	0.69	1.00
40	1.05	0.71	1.01
Model Checking			
iproto	1.17	0.65	1.15
leader	1.10	0.46	1.06
sieve	1.12	0.59	1.02
Left Last			
Cycle			
500	1.06	0.98	1.02
1000	1.05	0.92	0.96
1500	1.02	0.95	0.95
Pyramid			
500	1.09	0.98	1.00
1000	1.20	1.11	1.15
1500	0.98	1.02	1.03
Grid			
20	1.02	0.96	0.96
30	1.00	0.97	0.99
40	1.02	1.00	1.00
Model Checking			
iproto	1.09	1.05	1.07
leader	1.02	1.02	0.97
sieve	1.08	1.07	0.99

Appendix B

OpenRuleBench Tests

B.1 Local Scheduling

Table B.1: Running time ratios for local scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem (values higher than 1.00 mean that the optimization is better)

Programs					**DRA**	**DRA**	**DRE**	
		DRA	**DRE**	**DRS**	+	+	+	**All**
Data	**Par**				**DRE**	**DRS**	**DRS**	
Non-Cycle								
1000	250000	1.85	1.00	1.00	1.84	1.83	1.00	1.85
1000	500000	1.74	1.01	1.00	1.74	1.74	1.01	1.73
1000	50000	1.93	1.01	1.00	1.94	1.93	1.00	1.94
2000	1000000	1.86	1.00	1.00	1.85	1.85	1.00	1.87
2000	500000	1.92	1.01	1.00	1.91	1.90	1.01	1.92
Cycle								
1000	250000	1.19	1.01	1.00	1.51	1.50	1.01	1.51
1000	500000	1.49	1.00	0.99	1.48	1.47	1.00	1.50
1000	50000	1.40	1.03	1.01	1.39	1.41	1.02	1.43
2000	1000000	1.55	1.04	1.03	1.56	1.54	1.05	1.58
2000	500000	1.51	1.02	1.01	1.50	1.44	1.01	1.49

Table B.2: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem for Non-Cycle edges

Programs				**DRA**	**DRA**	**DRE**	
	DRA	**DRE**	**DRS**	+	+	+	**All**
Data Par				**DRE**	**DRS**	**DRS**	
1000 250000							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-992,740	0	-992,740	-992,740	-992,740
Alternatives	-1,740	0	0	-1,740	-1,740	0	-1,740
SCC Eval	-742	0	0	-742	-742	0	-742
1000 500000							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-998,069	0	-998,069	-998,069	-998,069
Alternatives	-1,493	0	0	-1,493	-1,493	0	-1,493
SCC Eval	-494	0	0	-494	-494	0	-494
1000 50000							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-934,620	0	-934,620	-934,620	-934,620
Alternatives	-1,917	0	0	-1,917	-1,917	0	-1917
SCC Eval	-932	0	0	-932	-932	0	-932
2000 1000000							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-3,987,905	0	-3,987,905	-3,987,905	-3,987,905
Alternatives	-3,511	0	0	-3,511	-3,511	0	-3.511
SCC Eval	-1,512	0	0	-1,512	-1,512	0	-1,512
2000 500000							
Tabled Nodes	0	0	0	0	0	0	0
Answers	0	0	-3,962,399	0	-3,962,399	-3,962,399	-3,962,399
Alternatives	-3,742	0	0	-3,742	-3,742	0	-3,742
SCC Eval	-1,747	0	0	-1,747	-1,747	0	-1,747

Table B.3: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem for Cycle edges

Programs		**DRA**	**DRE**	**DRS**	**DRA + DRE**	**DRA + DRS**	**DRE + DRS**	**All**
Data	**Par**							
1000	**250000**							
Tabled Nodes		0	0	0	0	0	0	0
Answers		0	0	-2,000,000	0	-2,000,000	-2,000,000	-2,000,000
Alternatives		-1,002	0	0	-1,002	-1,002	0	-1,002
SCC Eval		-1	0	0	-1	-1	0	-1
1000	**500000**							
Tabled Nodes		0	0	0	0	0	0	0
Answers		0	0	-2,000,000	0	-2,000,000	-2,000,000	-2,000,000
Alternatives		-1,002	0	0	-1,002	-1,002	0	-1,002
SCC Eval		-1	0	0	-1	-1	0	-1
1000	**50000**							
Tabled Nodes		0	0	0	0	0	0	0
Answers		0	0	-2,998,566	0	-2,998,566	-2,998,566	-2,998,566
Alternatives		-2,002	0	0	-2,002	-2,002	0	-2,002
SCC Eval		-1	0	0	-1	-1	0	-1
2000	**1000000**							
Tabled Nodes		0	0	0	0	0	0	0
Answers		0	0	-8,000,000	0	-8,000,000	-8,000,000	-8,000,000
Alternatives		-2,002	0	0	-2,002	-2,002	0	-2,002
SCC Eval		-1	0	0	-1	-1	0	-1
2000	**500000**							
Tabled Nodes		0	0	0	0	0	0	0
Answers		0	0	-8,000,000	0	-8,000,000	-8,000,000	-8,000,000
Alternatives		-2,002	0	0	-2,002	-2,002	0	-2,002
SCC Eval		-1	0	0	-1	-1	0	-1

B.2 Batched Scheduling

Table B.4: Running time ratios for batched scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem (values higher than 1.00 mean that the optimization is better)

Programs		**DRA**	**DRE**	**DRA + DRE**
Data	**Par**			
Non-Cycle				
1000	250000	1.85	1.01	1.85
1000	500000	1.69	1.01	1.74
1000	50000	1.96	0.97	1.93
2000	1000000	1.82	1.01	1.86
2000	500000	1.89	1.01	1.91
Cycle				
1000	250000	1.52	1.07	1.52
1000	500000	1.44	1.01	1.44
1000	50000	1.54	1.10	1.53
2000	1000000	1.56	1.10	1.56
2000	500000	1.63	1.16	1.64

Table B.5: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem for Non-Cycle edges

Programs		**DRA**	**DRE**	**DRA + DRE**
Data	**Par**			
1000	**250000**			
Tabled Nodes		0	0	0
Alternatives		-998	0	-998
SCC Eval		-742	0	-742
1000	**500000**			
Tabled Nodes		0	0	0
Alternatives		-999	0	-999
SCC Eval		-494	0	-494
1000	**50000**			
Tabled Nodes		0	0	0
Alternatives		-985	0	-985
SCC Eval		-932	0	-932
2000	**1000000**			
Tabled Nodes		0	0	0
Alternatives		-1,999	0	-1,999
SCC Eval		-1,512	0	-1,512
2000	**500000**			
Tabled Nodes		0	0	0
Alternatives		-2,001	0	-2,001
SCC Eval		-1	0	-1

Table B.6: Statistics for local scheduling comparing standard linear tabling against the several optimizations using the transitive closure with no query bindings (free-free version) OpenRuleBench problem for Cycle edges

Programs		**DRA**	**DRE**	**DRA + DRE**
Data	**Par**			
1000	**250000**			
Tabled Nodes		0	0	0
Alternatives		-1,001	0	-1,001
SCC Eval		-1	0	-1
1000	**500000**			
Tabled Nodes		0	0	0
Alternatives		-1,001	0	-1,001
SCC Eval		-1	0	-1
1000	**50000**			
Tabled Nodes		0	0	0
Alternatives		-1,001	0	-1,001
SCC Eval		-1	0	-1
2000	**1000000**			
Tabled Nodes		0	0	0
Alternatives		-2,001	0	-2,001
SCC Eval		-1	0	-1
2000	**500000**			
Tabled Nodes		0	0	0
Alternatives		-2,001	0	-2,001
SCC Eval		-1	0	-1

B.3 Comparison with YapTab

Table B.7: Running time ratios for local and batched scheduling comparing standard linear tabling against YapTab and the best linear optimization using the transitive closure with no query bindings (free-free version) OpenRuleBench problem for Non-Cycle and Cycle edges

Programs		**Local Scheduling**		**Batched Scheduling**	
Data	**Par**	**YapTab**	**Best Linear (Opt)**	**YapTab**	**Best Linear (Opt)**
Non-Cycle					
1000	250000	2.00	1.85 (DRA)	2.00	1.85 (DRA)
1000	500000	2.01	1.74 (DRA+DRE)	1.91	1.74 (DRA+DRE)
1000	50000	1.99	1.44 (All)	2.02	1.96 (DRA)
2000	1000000	1.97	1.87 (All)	2.03	1.86 (DRA+DRE)
2000	500000	2.01	1.92 (DRA)	2.01	1.90 (DRA+DRE)
Cycle					
1000	250000	1.84	1.51 (DRA+DRE)	2.21	1.52 (DRA)
1000	500000	1.89	1.50 (All)	2.07	1.44 (DRA)
1000	50000	2.20	1.30 (All)	2.24	1.54 (DRA)
2000	1000000	1.91	1.58 (All)	2.29	1.56 (DRA+DRE)
2000	500000	1.82	1.51 (DRA)	2.40	1.64 (DRA+DRE)

References

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

[2] K. Apt and M. van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM*, 29(3):841–862, 1982.

[3] M. Areias and R. Rocha. An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages, PADL'2010*, number 5937 in LNCS, pages 279–293, Madrid, Spain, January 2010. Springer-Verlag.

[4] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, 1990.

[5] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[6] P. Chico, M. Carro, M. V. Hermenegildo, C. Silva, and R. Rocha. An Improved Continuation Call-Based Implementation of Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 4902 in LNCS, pages 197–213. Springer-Verlag, 2008.

[7] K. Clark. Predicate Logic as a computational formalism. Research monograph, Imperial College, December 1979.

[8] 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.

[9] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in LNCS, pages 21–35. Springer-Verlag, 1998.

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

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

[12] Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer-Verlag, 2001.

[13] R. Karlsson. *A High Performance OR-parallel Prolog System.* PhD thesis, The Royal Institute of Technology, 1992.

[14] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[15] R. Kowalski and Donald Kuehner. Linear Resolution with Selection Function. In *Artificial Intelligence 2*, pages 227–260. North-Holland, 1971.

[16] S. Liang, P.Fodor, H. Wan, and M.Kifer. OpenRuleBench: An Analysis of the Performance of Rule Engines. In *Internacional World Wide Web Conference Committee*, Madrid, Spain, April 2009. ACM Press.

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

[18] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[19] Richard A. O'Keefe. *The Craft of Prolog.* The MIT Press, 1990.

[20] C. Pusch. Verification of compiler correctness for the WAM. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics(TPHOL-96)*, number 1125 in LNCS, Turku, Finland, 1996. Springer-Verlag.

[21] 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*, pages 687–711. The MIT Press, 1995.

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

[23] R. Ramesh and W. Chen. Implementation of Tabled Evaluation with Delaying in Prolog. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):559–574, 1997.

[24] P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, number 1265 in LNCS, pages 431–441. Springer-Verlag, 1997.

[25] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[26] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs.* PhD thesis, Department of Computer Science, University of Porto, 2001.

[27] R. Rocha, C. Silva, and R. Lopes. Implementation of Suspension-Based Tabling in Prolog using External Primitives. In *Local Proceedings of the 13th Portuguese Conference on Artificial Intelligence*, pages 11–22, 2007.

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

[29] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[30] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

[31] V. Santos Costa, K. Sagonas, and R. Lopes. Demand-Driven Indexing of Prolog Clauses. In *International Conference on Logic Programming*, number 4670 in LNCS, pages 395–409. Springer-Verlag, 2007.

[32] Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer-Verlag, 2006.

[33] T. Swift and R.Marques. Concurrent and Local Evaluation of Normal Programs. In *International Conference on Logic Programming*, number 5366 in LNCS, pages 206–222. Springer-Verlag, 2008.

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

[35] Nilsson Ulf and Jan M. *Logic, Programming and Prolog.* John Wiley and Sons, Sweden, 1995.

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

[37] D. H. D. Warren. Implementation of Prolog. In *5th International Conference and Symposium on Logic Programming*, 1988.

[38] Neng-Fa Zhou, T. Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.

[39] 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*, number 1753 in LNCS, pages 109–123. Springer-Verlag, 2000.