Flávio Manuel Fernandes Cruz

Call Subsumption Mechanisms for Tabled Logic Programs

Universidade do Porto

Faculdade de Engenharia

FEUP

Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Junho de 2010

Flávio Manuel Fernandes Cruz

Call Subsumption Mechanisms for Tabled Logic Programs

Dissertação submetida à Faculdade de Engenharia da Universidade do Porto como parte dos requisitos para a obtenção do grau de Mestre em Engenharia Informática e de Computação

Supervisor: Ricardo Rocha

Departamento de Engenharia Informática
Faculdade de Engenharia da Universidade do Porto
Junho de 2010

To my parents

Acknowledgments

I would like to thank my supervisor, Prof. Ricardo Rocha, for the help and encouragement during the development of this thesis. He was always there to listen and to give advice. I have certainly become a better researcher because of him.

To João Santos, João Raimundo, José Vieira and Miguel Areias for the excellent work environment and companionship.

To all my friends from FEUP, for their friendship and the great moments we have spent together in the last five years.

Finally, I would like to thank my parents and sisters, for their unconditional love and support, and to Joana, for her affection and understanding.

Flávio Cruz

June 2010

Abstract

Tabling is a particularly successful resolution mechanism that overcomes some limitations of the SLD resolution method found in Prolog systems, namely, in dealing with recursion and redundant sub-computations. In tabling, first calls to tabled subgoals are evaluated through program resolution, while *similar calls* are evaluated by consuming answers stored in the table space by the corresponding similar subgoal. In general, we can distinguish between two main approaches to determine if a subgoal A is similar to a subgoal B: *variant-based tabling* and *subsumption-based tabling*. In variant-based tabling, A is similar to B if they are the same up to variable renaming. In subsumption-based tabling, A is similar to B when A is subsumed by B (or B subsumes A). This stems from a simple principle: if A is subsumed by B and S_A and S_B are the respective answer sets, then $S_A \subseteq S_B$. While subsumption-based tabling can yield superior time performance by allowing greater answer reuse, its efficient implementation is harder than variant-based tabling, which makes tabling engines with variant checks much more popular in the logic community.

This thesis first addresses the porting and integration of the *Time-Stamped Tries* mechanism from the SLG-WAM tabling engine into the YapTab tabling engine. Our performance results show that our integration efforts were successful, with comparable speedups when using subsumptive-tabling against variant-tabling.

In the second part of this thesis we present the design, implementation, and evaluation of a novel extension based on subsumption-based tabling called *Retroactive Call Subsumption* (RCS). RCS overcomes some limitations of traditional call subsumption, namely, the fact that the call order of the subgoals can greatly affect its success and applicability. RCS allows full sharing of answers, independently of the order they are called by selectively pruning and restarting the evaluation of subsumed subgoals. Our results show considerable gains for programs that can take advantage of RCS, while programs that do not benefit from it show a small overhead using the new mechanisms.

Resumo

A tabulação é um método de resolução particularmente bem sucedido que resolve algumas das limitações do método de avaliação SLD encontrado em sistemas Prolog, no tratamento de computações recursivas e/ou redundantes. Na tabulação, as primeiras chamadas a subgolos tabelados são avaliadas normalmente através da execução do código do programa, enquanto que *chamadas similares* são avaliadas através do consumo das respostas geradas pelo subgolo similar correspondente. Em geral, podemos distinguir entre duas formas de determinar se um subgolo A é similar a um subgolo B: *tabulação por variantes* e *tabulação por subsumpção*. Na tabulação por variantes, A é similar a B quando eles são iguais por renomeação das variáveis. Na tabulação por subsumpção, A é similar a B quando A é mais específico do que B (ou B é mais geral do que A). Isto acontece pelo simpes princípio de que se A é mais específico do que B e S_A e S_B são os respectivos conjuntos de respostas, então $S_A \subseteq S_B$. Embora a tabulação por subsumpção consiga atingir maiores ganhos em termos do tempo de execução, devido à maior partilha de respostas, a implementação eficiente dos mecanismos necessários para seu suporte é bastante mais difícil em comparação com tabulação por variantes, o que faz com que este último seja bastante mais popular entre os motores de tabulação disponíveis.

Esta tese descreve a migração e integração do mecanismo de *Time-Stamped Tries* do motor de tabulação SLG-WAM no motor de tabulação YapTab. Os resultados obtidos mostram que os nossos esforços de integração foram bem sucedidos, com desempenhos comparavéis aos da SLG-WAM na execução entre tabulação por variantes e tabulação por subsumpção.

Na segunda parte desta tese apresenta-se o desenho, implementação e avaliação de uma nova extensão baseada na tabulação por subsumpção chamada *Tabulação por Subsumpção Retroactiva* (TSR). A TSR resolve algumas limitações da tabulação por subsumpção tradicional, nomeadamente, o facto de a ordem da chamada dos subgolos poder afectar o seu sucesso e aplicação. A TSR permite uma partilha completa e bidireccional de respostas entre subgolos, independentemente da sua ordem de chamada através do corte da avaliação dos golos mais específicos. Os nossos resultados mostram ganhos consideráveis para os programas que conseguem tirar partido do novo mecanismo, enquanto que o custo associado aos programas que dele não conseguem beneficiar é quase insignificante.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic programming is a very high level programming paradigm that allows the programmer to focus on the declarative aspects of the problem, instead of describing the specific steps needed to solve it. Arguably, the Prolog language is the most popular logic programming language. Part of the success of the Prolog language can be attributed to the development of a fast and very efficient sequential machine called the *Warren's Abstract Machine* (WAM) [War83]. The advances in WAM technology and optimization techniques enabled Prolog to be applied in real world problems in a wide range of fields such as Artificial Intelligence, Natural Language Processing, Machine Learning, Knowledge Based Systems, Database Management, or Expert Systems.

While the declarative aspect of Prolog is based on mathematical logic and predicate calculus, its operational semantics is based one a relatively simple refutation strategy called *Selective Linear Definite* (SLD) [Llo87], which is a well defined evaluation method for logic programs that is particularly well suited to stack based machines. Furthermore, Prolog defines a few extra-logical constructs, such as the *cut operator* that give the programmer more control over the evaluation. Both the SLD operational semantics and the cut operator make the programmer more aware of the actual evaluation process in detriment to the declarative aspect of the language that is naturally non-deterministic. It is possible to exploit these deterministic rules to speedup execution, and thus, for example solve problems related to redundant sub-computations. Notwithstanding, standard Prolog has still some deficiencies. For instance, writing left-recursive programs can lead to infinite loops.

There have been some attempts in making Prolog less prone to problems related to recursion and redundant sub-computations, in order to make the language more expressive and closer to its mathematical logic foundations. One of these attempts, which is particularly successful, is called *tabling* (or *tabulation* or *memoization* [Mic68]). The tabling technique stems from one simple idea: store intermediate answers in a place called the *table space* and reuse those answers when a *similar call* appears during the resolution process. Tabling refines the SLD resolution method by distinguishing between first calls to *tabled subgoals*, which are evaluated as usual through *program resolution*, and similar calls to tabled subgoals, which are evaluated through *answer resolution*, i.e., by consuming answers that are being stored in the table space by the corresponding similar subgoal, instead of being re-evaluated against the program clauses. Tabled evaluation is able to reduce the search space, avoid looping, and has better termination properties than traditional SLD resolution [CW96]. The advantages of tabling have lead to its application in fields such as Deductive Databases [SSW94], Program Analysis [RRS$^+$00], Knowledge Based Systems [YK00], Inductive Logic Programming [RFS05], and Model Checking [RRS$^+$00].

In tabling, *call similarity* determines if a subgoal A is similar to a subgoal B, in other words, whether A will generate its own answers or will consume answers from B. In general, we can distinguish between two main approaches for call similarity:

- *Variant-based tabling*: A and B are variants if they can be made identical through variable renaming as proposed by Bachmair *et al* [BCR93]. For example, subgoals $p(X, 1, Y)$ and $p(Y, 1, Z)$ are *variants*, because both can be transformed into $p(VAR_0, 1, VAR_1)$;

- *Subsumption-based tabling* or *tabling by call subsumption*: Subgoal A is considered similar to B if A is *subsumed* by B (or B *subsumes* A), i.e., if A is more specific than B (or an instance of). For example, subgoal $p(X, 1, 2)$ is subsumed by subgoal $p(Y, 1, Z)$ because there is a substitution $\{Y = X, Z = 2\}$ that makes $p(X, 1, 2)$ an instance of $p(Y, 1, Z)$. Tabling by call subsumption is based on the principle that if A is subsumed by B and S_A and S_B are the respective answer sets, then $S_A \subseteq S_B$.

In general, subsumption-based tabling has the following advantages over variant-tabling: superior time performance, because less program resolution is required; and less space requirements, as it allows greater reuse of answers, since the answer sets for the subsumed subgoals are not stored. However, the mechanisms to efficiently support subsumption-based tabling are more complex and harder to implement, which makes the variant-based tabling approach more popular within the available tabling systems, such as YapTab [RSS00], B-Prolog [ZSYY00], and ALS-Prolog [GG01]. To the best of our knowledge, the SLG-WAM [SS98] engine from XSB Prolog is the sole tabling system that supports subsumption-based tabling, initially by using an organization of the table space called *Dynamic Threaded Sequential Automata (DTSA)* [RRR96], and later by using an alternative design called *Time-Stamped Tries (TST)* [JRRR99]. While the DTSA design was more efficient, it had bigger space requirements and thus is was replaced by the simpler TST approach which uses far less memory.

1.1 Thesis Purpose

In this thesis we address the design, implementation, integration and evaluation of two subsumption-based engines built on top of YapTab [RSS00], the tabling system that is part of Yap Prolog. For the first engine, we reused and integrated the *Time-Stamped Tries* (TST) approach from XSB Prolog into Yap Prolog. We studied how subsumption-based and variant-based tabling were seamlessly integrated into the SLG-WAM engine and we attempted to reuse most of the original code and data structures when integrating these new mechanisms into YapTab. Consequently, we made minimal modifications to the YapTab engine that enabled it to support a mix of variant and subsumptive subgoals on the same program. Our performance results show that our integration efforts were successful, with comparable speedups to the SLG-WAM when using subsumptive-tabling against variant-tabling.

For the second system, we designed a novel extension for subsumptive-tabling called *Retroactive Call Subsumption* (RCS). This extension attempts to solve one major problem in traditional call subsumption: the order in which subgoals are called during a particular evaluation can greatly affect the success and applicability of the call by subsumption technique. For example, if more specific subgoals are called before the more general subgoal, no reuse will be employed, while if the more general subgoal is called first, reuse will happen. The RCS extends the original TST design by allowing full sharing of answers, independently of the order they are called. The basic idea is to selectively prune and restart the evaluation of generator subgoals that are subsumed by a new called subgoal in order to reuse the answers from the subsuming subgoal, instead of continuing to generate their own answers.

To implement retroactive-based tabling we developed a few novel ideas: (1) a novel algorithm to efficiently traverse the table space data structures and retrieve the running *instances* of a subgoal; (2) a novel table space organization, based on the ideas of the *common global trie* proposal [CR08], where answers are represented only once; and (3) a new evaluation strategy capable of pruning and transforming generator nodes into consumer nodes.

Our results show that the overhead of the new mechanisms for RCS support are low enough in programs that do not benefit from it, which, combined with considerable gains for programs that can take advantage of them, validates this new evaluation technique. With this in mind, we argue that Retroactive Call Subsumption makes tabling more adapted and useful for practical applications and is another great functionality in the programmer's toolbox for writing tabled logic programs.

1.2 Thesis Outline

In the following list we describe each chapter of this thesis.

Chapter 1: Introduction. Is this chapter.

Chapter 2: Logic Programming and Tabling. Provides a overview of the main topics of this thesis. The subjects discussed are logic programming, Prolog, and tabling for logic programs. A brief description of the YapTab and SLG-WAM tabling engines is also presented.

Chapter 3: Table Space Organization. Describes the table space organization for both variant and subsumption-based tabling engines. We start by describing the variant table space for both SLG-WAM and YapTab systems. We then give a brief overview about the table space organization for the DTSA and TST techniques that implement tabling with subsumptive checks.

Chapter 4: Time Stamped Tries. Throughly presents the Time Stamped Tries approach to subsumption-based tabling. First, we describe the algorithm used to detect subsuming subgoals. Next, we give a detailed description of the data structures used in the table space that are used to speedup the identification of relevant answers for subsumed subgoals. Finally, we focus on the modifications we have made to the YapTab tabling engine in order to support tabling by call subsumption based on the TST approach.

Chapter 5: Retroactive Call Subsumption. We start with the motivations behind RCS, by showing the shortcomings of pure subsumption-based tabling.

We next describe the rules for the new mechanism and the problems that arise when pruning execution branches. Finally, we discuss the novel table space organization called *Single Time Stamped Trie* (STST) and then we throughly describe the new algorithm developed to find executing subsumed subgoals of a subgoal on the table space.

Chapter 6: Experimental Results. This chapter first presents the experimental results we achieved with the new YapTab engine that reuses the TST approach and how it compares to the SLG-WAM. We also make a space analysis comparison between call subsumption and variant-based tabling. Next, we present and discuss the overhead of the RCS mechanism on programs that do not benefit from it and the speedups we have achieved for programs that can take advantage of it. Finally, we present an analysis of the STST table organization by experimenting with programs that stress the nature of this table space organization.

Chapter 7: Conclusions. Summarizes the work, enumerates the contributions and suggests directions for future work.

Chapter 2

Logic Programming and Tabling

The purpose of this chapter is to give an overview of the research areas involved in this dissertation. First we explain the fundamental ideas of logic programming and Prolog; next, the main concepts behind tabling evaluation are described, focusing on the execution rules and strategies; finally, we introduce the Yap and XSB Prolog systems, focusing on the tabling engines designed for both systems.

2.1 Logic Programming

Logic programming presents a declarative style of programming based on mathematical logic and the predicate calculus. It is a very high level programming paradigm that allows the programmer to focus on the problem at hand, leaving the steps on *how* to solve the problem to the computer.

In its purest form, logic programming is solely based on Horn Clause Logic [Llo87], a subset of First Order Logic. Programming in logic can be viewed as a two step process: (1) first, the theory is formulated as logic clauses, next (2) we use this theory to search for alternative ways in which an arbitrary query is satisfied.

Logic programming is often mentioned to include the following advantages [Car90]:

- **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. In Prolog, for instance, 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.

2.1.1 Logic Programs

A logic program is composed by a set of Horn clauses. Each clause is a disjunction of literals and contains at most one positive literal. Horn clauses are usually written as

$$L_1, ..., L_n \Longrightarrow L(\equiv \neg L_1 \vee ... \neg L_n \vee L)$$

or

$$L_1, ..., L_n(\equiv \neg L_1 \vee ... \neg L_n)$$

where $n >= 0$ and L is the only positive literal.

A Horn clause that has exactly one positive literal is called a definitive clause; in the Prolog language it is usually called a *rule*. A Horn clause without a positive literal is called a *goal*.

Using Prolog's notation, one can write *rules* in the form

$$L : -L_1, ..., Ln.$$

Usually, L is called the *head* of the *rule* and $L_1, ..., L_n$ the *body* of the *rule*, where each L_i is called a subgoal. A logical *fact* is a special *rule* where the *body* is replaced by the *true* symbol:

$$L.$$

Goals are *rules* without the *head* component and are also named as *queries*.

Each literal in a Horn clause has the form $f(t_1, ..., t_n)$, where f is the *functor* symbol and each t_i are *terms*. A term can be a *constant* (or *atom*), a *variable* or a *compound* term. Compound terms follow the functor structure, recursively. Variables are assumed to be universally quantified and have the following major characteristics:

- Variables are logical variables that can instantiated only once.
- Variables are untyped until instantiated.
- Variables are instantiated via *unification*, a pattern matching operation that finds the most general common instance of two data objects.

A sequence of clauses with the same functor in the head form a *predicate*. The ordering of these clauses can have some implications depending on the resolution semantics of the underlying language. Prolog for instance, uses a top-down resolution mechanism known as *SLD* (Selective Linear Definite) resolution [Llo87].

SLD starts by matching the first subgoal query to the first clause of the respective predicate, generating a new query using the body of the clause, which is added to the remaining query subgoals. During this process a finite set of pairs θ called *substitution* is built. Each pair has the form $X = t$, where X is a variable and t is a term. No variable in the left-hand side of a pair appears on the right-hand side and no two pairs have the same variable as left-hand side [SS94]. When the clause body is reused as query, all the variables present in the terms are replaced using the set θ. If unification fails, the next clause of the predicate is tried, using a mechanism called *backtracking*. This recursive computation fails when there are any more clauses left to try. It succeeds when the subgoal query is empty.

The resolution process is fundamentally non-deterministic and can be viewed as a search within a tree. SLD does not force any specific search strategy for exploring the tree. Prolog for example, uses a depth-first, one branch at a time search.

2.1.2 Prolog and the WAM

Prolog is one of the first logic programming languages and arguably the most successful. The first implementation of Prolog was Marseille Prolog, developed in 1972 by Alain Colmerauer and Robert Kowalski [Kow74].

The use of Prolog as a practical and efficient language was made viable by David Warren in 1977, when he built a compiler that could compete with other languages like Lisp [WPP77]. Then in 1983, David Warren formulated an abstract machine known as WAM (*Warren Abstract Machine*) [War83] that is still widely used in modern Prolog implementations.

2.1.2.1 Prolog

Prolog follows the semantics of the SLD resolution through a depth first search strategy. It starts by choosing the top-most clauses of the predicate and the subgoals are solved within a left-to-right fashion.

```
factorial(0,1) :- !.
factorial(N,R) :-
  N > 0,
  N1 is N - 1,
  factorial(N1,R1),
  R is N * R1.
```

Figure 2.1: Factorial function in Prolog.

For illustration purposes, in Figure 2.1 we define the predicate `factorial/2` that computes the factorial of a given number. This predicate has arity of 2, where the first argument is an *input* argument and the second argument an *output* argument. Factorial is composed of two clauses, the first represents the factorial base case (factorial of 0 is 1) and the second represents the recursive relation. The second clause first checks if the input number is positive, to discard non-positive numbers, then computes $N-1$ and recursively calls factorial to compute the value of $factorial(N-1)$. Finally, the output argument R is then unified to $N * factorial(N-1)$. The first clause uses the *cut* operator (`!`) that tells the Prolog engine to not explore alternative clauses, i.e., the factorial of 0 is not to be computed using the recursive call defined on the second clause. Once Prolog finds the first answer (Figure 2.2), the cut control operator disables further alternatives, completing the depth first search in the tree.

The cut operator is not the only special instruction in Prolog, more built-in predicates are also available:

- **Meta-logical predicates**: inquire the state of the computation and manipulate

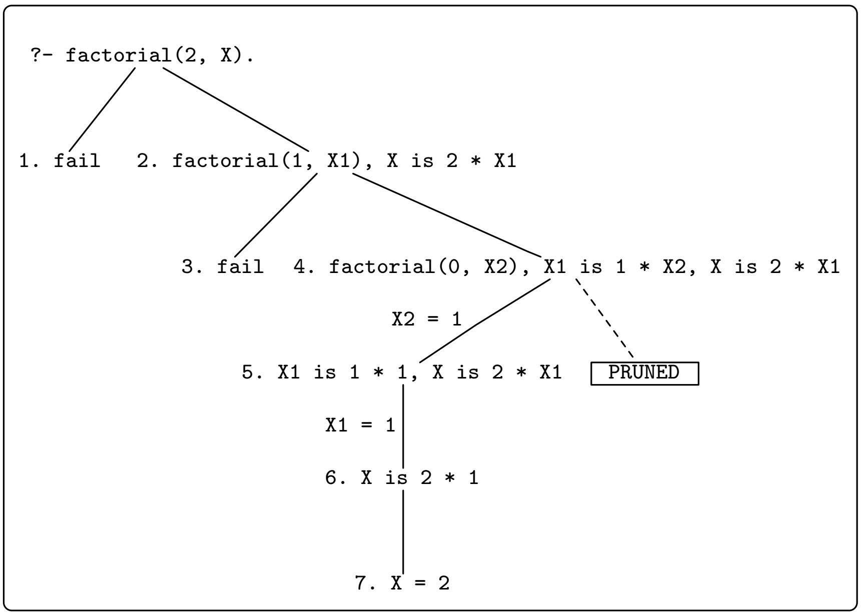

?- factorial(2, X).
1. fail
2. factorial(1, X1), X is 2 * X1
3. fail
4. factorial(0, X2), X1 is 1 * X2, X is 2 * X1
X2 = 1
5. X1 is 1 * 1, X is 2 * X1
PRUNED
X1 = 1
6. X is 2 * 1
7. X = 2

Figure 2.2: Factorial search tree.

terms.

- **Extra-logical predicates**: manipulate the Prolog database, adding or removing clauses from the program being executed. Input/Output operators are another example of extra-logical predicates.

- **Other predicates**: predicates to perform arithmetic operations, to compare terms, to support debugging, etc.

These special operators make programming more practical and useful in real world applications.

2.1.2.2 WAM

The *Warren Abstract Machine* (WAM) is a stack-based architecture with various data areas, registers, and low level instructions that can be efficiently executed, manipulated and optimized. A simplified layout is presented in Figure 2.3.

In terms of execution stacks, the WAM defines the following:

- **PDL**: a push down list used by the unification process.
- **Trail**: stores the addresses of the variables that must be reset when backtracking.
- **Stack**: stores *environment* and *choice point* frames. Environments track the flow control in a program and consist of: the stack address of the previous environment; a pointer to the next instruction to execute upon return of the invoked clause; and a set of *permanent variables* [1] as a sequence of cells.

  Choice points store open alternatives which are used to restore the state of the computation when backtracking. A pointer to the instruction for the next alternative is stored in case the current execution branch fails. A choice point is created when there are more than one alternative for a subgoal call. We pop the choice point from the stack when the last alternative clause is attempted.
- **Heap**: array of data cells used to store variables and compound terms that cannot be stored in the stack.
- **Code Area**: contains the compiled instructions.

For the registers, WAM defines the following:

- **P**: points to the current WAM instruction.
- **CP**: stores the value of **P** before the current invoked call and it is used to restore the execution point.
- **TR**: points to the top of the trail stack.
- **E**: points to the current active environment.
- **B**: the active choice point.
- **B0**: the choice point to return to upon backtracking over a cut.
- **H**: points to the top of the heap stack

[1] A permanent variable is a variable which occurs in multiple body subgoals and must be preserved between calls.

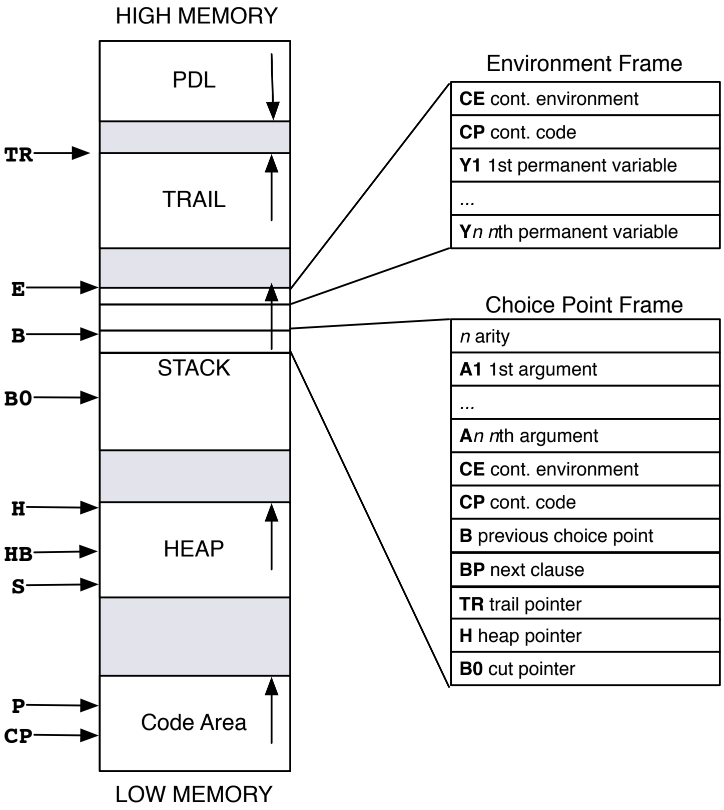


Figure 2.3: WAM memory layout, frames and registers.

- **HB**: marks the value of the register **H** at the time of the latest choice point. It is used to determine *conditional* variable bindings that affect variables existing before the creation of the choice point.

- **S**: used during the unification of compound terms.

WAM instructions can be grouped into four main groups: choice point instructions to manipulate choice points; control instructions to manage environments and control the execution flow; unification instructions that implement specialized versions of the unification algorithm; and indexing instructions to efficiently determine which clauses unify with a given subgoal call.

The WAM being a complex topic has complete books dedicated to explaining its intricacies. An example is the *Warren's Abstract Machine – A Tutorial Reconstruction* written by H. Aït-Kaci [AK91].

2.2 Tabling

Despite Prolog's declarativeness and expressiveness, the past few years have seen wide efforts at solving shortcomings that arise when using SLD resolution. One proposal that has gained popularity is *tabling* or *tabulation* [CW96]. In comparison to the traditional SLD resolution method, tabling can reduce the search space to cut redundant computations, avoids looping and has better termination properties [TS86].

In a nutshell, tabling is a refinement of the SLD resolution that consists in storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears in the resolution process. The use of tabling enables the programmer to write more expressive, but still valid, logical programs.

One classical example that is used to demonstrate the advantages of using tabling is presented in Figure 2.4. This program describes the predicate `path/2` that computes reachability between two nodes on a directed graph. Connections are established as facts using the `edge/2` predicate.

```
:- table path/2.

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

edge(a,b).
edge(b,a).
```

Figure 2.4: The `path` program.

If we tried to evaluate the query goal `path(X,Z)`, traditional Prolog would enter an infinite loop (Figure 2.5) because `edge/2` facts define a cyclic graph and the first clause of `path/2` is right recursive, leading to a repeated call.

2.2.1 Tabled Evaluation

In this new method of evaluation, when a tabled subgoal is first called, a new entry is allocated on the *table space*. Table entries are used to store subgoal calls but they also store answers found during evaluation. Each time a tabled subgoal is called, we know if it is a repeated call by inspecting the table space. Nodes in the search space

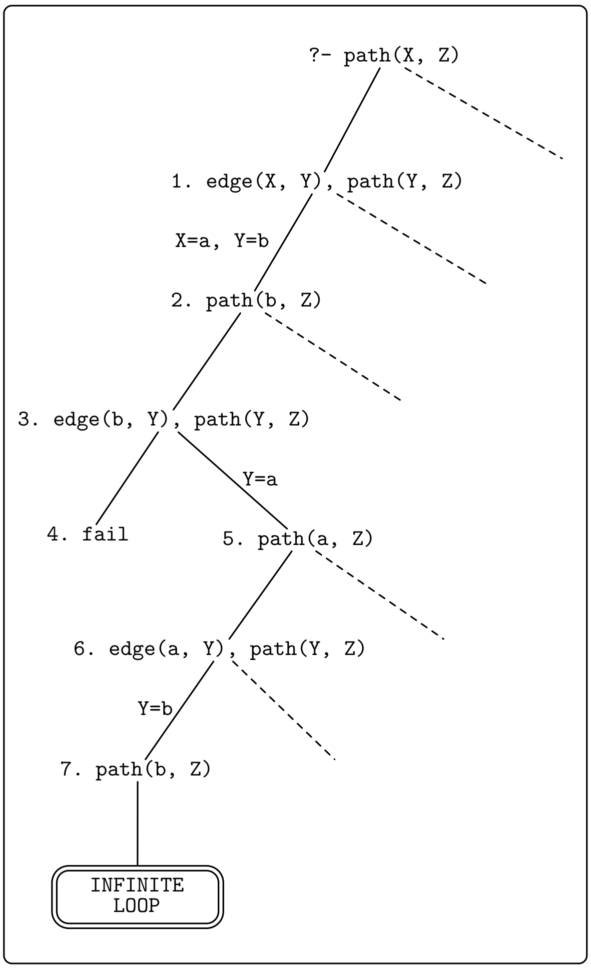


Figure 2.5: Infinite loop evaluating `path(X, Z)`.

can thus be classified as: *generator nodes*, if they are being called for the first time; *consumer nodes* if they are repeated calls; or *interior nodes* if they are non-tabled subgoals. Generator nodes are matched against the predicate clauses as usual but consumer nodes are not, instead they consume answers stored in the table space from the respective subgoal.

In Figure 2.6 we depict the tabled evaluation of the query goal `path(X,Z)`. Generator nodes are represented by rectangles with double lines and consumer nodes by simple rectangles. Note that we need to declare `path/2` as tabled using the `table` directive.

Tabled evaluation starts by inserting a new entry in the table space and by allocating a generator node to represent `path(X,Z)` (step 1). Like SLD resolution, `path(X,Z)` is then resolved against the first `path/2` clause (step 2). The goal `edge(X,Y)` is not tabled and is resolved as usual. We use the first `edge/2` clause with {`X = a, Y = b`} and these values are carried to `path(b,Z)` (step 3). This goal is not yet in the table space, hence we add a new entry for it.

Goal `path(b,Z)` is then resolved against the first clause of `path/2` (step 4). Ñext, `edge(b,Y)` fails against the first clause but succeeds with {`Y = a`} for the second clause. A new tabled subgoal `path(a,Z)` is registered in the tabled space (step 6) and resolved against the first clause of `path/2` (step 7). This time the `edge/2` subgoal matches with the first clause with {`Y = b`}. This originates a repeated tabled subgoal call to `path(b,Z)` and the first consumer node is allocated (step 8). As we have no answers for `path(b,Z)` in the table space, the current evaluation point is *suspended.* Later on, this node can be resumed to consume new answers.

Next, we backtrack to node 7 and try the second `edge/2` clause, but resolution fails (step 9). We backtrack again, this time to node 6 to try the second clause of `path/2` (step 10). Here `edge(a, Z)` is resolved against the first clause of `edge/2` and the answer {`Z = b`} is found for the subgoal `path(a,Z)` (step 11). This answer is stored in the table space and forward execution is made, propagating the binding {`Z = b`} to `path(b,Z)`, and a first answer to this subgoal is also found and stored in the table space (step 12). We continue forward execution and the binding is once again propagated, this time to node 3 finding an answer to `path(X,Z)` and to the query subgoal, {`X = a, Z = b`} (step 13).

If the user asks for more answers, the computation returns to node 10 to try the second clause of `edge/2`, but it fails (step 14). The process backtracks to node 6 but at this node there are no more clauses left to try. Moreover, we can not complete the subgoal `path(a,Z)` because it depends on a older subgoal (node 3 is the generator node for the consumer node 8 under it), thus we backtrack to node 3.

At node 3, the second clause of `path/2` is tried (step 15) and the first clause of `edge/2` fails (step 16), but the second succeeds with a new answer for `path(b,Z)`, {`Z = a`} (step 17). Again, we propagate variable bindings in step 18, generating a new answer to subgoal `path(X,Z)`, {`X = a, Z = a`}.

We go back to node 3, where no more clauses are available, but now completion can be safely attempted because all consumers in lower branches do not depend on any generator node older than node 3. Node 3 is called, by definition, a *leader node* and the branch of nodes below it form a *Strongly Connected Component* (SCC) [Tar72].

However, by inspecting the execution tree, we can see that node 8 has two unconsumed answers. We thus resume the computation at node 8 to consume the answer {`Z = b`}

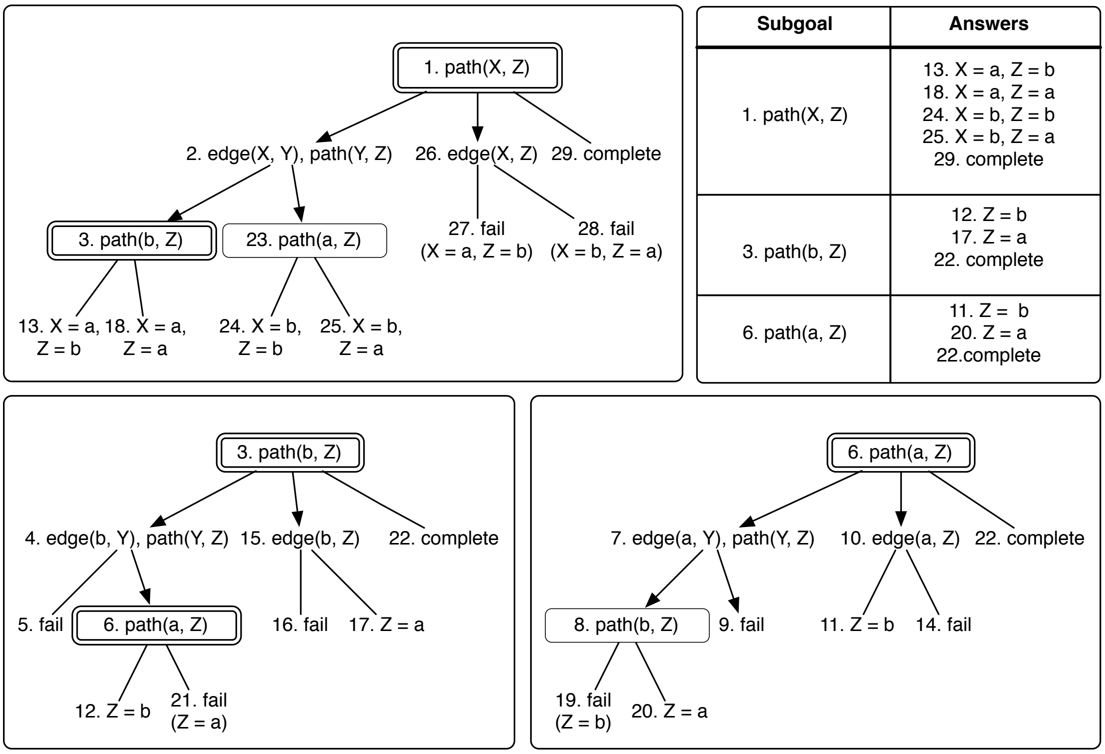


Figure 2.6: Tabled evaluation of `path(X,Z)`.

and forward it to subgoal `path(a,Z)` at node 6 (step 19). Here, we note that it is a repeated answer to this subgoal by checking the table space, and thus we fail (step 19). Failing repeated answers is crucial to avoid unnecessary computations and sometimes looping.

Then, we fetch the next available answer, {`Z = a`}, that is propagated to node 6, generating a new answer to `path(a,Z)` (step 20). This binding is once again propagated, now for node 3 but it is a repeated answer, and thus the computation fails (step 21). With no more unconsumed answers, we return back to node 3 to re-attempt completion. This time, no consumers have unconsumed answers and we can safely complete all the subgoals in the current SCC (step 22). Subgoals `path(b,Z)` (node 3) and `path(a,Z)` (node 6) are marked as **complete** in the table space and no new answers are accepted.

Next, we backtrack to node 2 to try the second `edge/2` clause. A new consumer node is allocated (step 23) and answers can be promptly consumed from the table space as the subgoal `path(a,Z)` is already completed. The retrieved answers in step 23 are propagated to node 1 and new answers are generated (steps 24 and 25).

We backtrack to node 1 to try the second `path/2` clause (step 26). Resolution succeeds for both `edge/2` clauses but as the newly found answers are repeated we fail for both cases (steps 27 and 28). The process backtracks again to node 1 and with no more clauses to try, we attempt completion. As there are no consumer nodes, completion is done (step 29) and computation terminates successfully.

From the described example we can summarize four main operations needed to support tabled evaluation:

- The *tabled subgoal call* operation represents a call to a tabled subgoal. If the subgoal is already in the table space, it creates a new consumer node to consume answers. If it is a new subgoal call it creates a new generator node and adds a new entry to the table space. Each new entry is initialized with an empty set S that will contain the answers for the subgoal.
- The *new answer* operation adds a new answer s to the table space. If the answer is repeated the operation fails, otherwise a new answer set S' for the subgoal is generated: $S' \equiv S \cup s$.
- The *answer resolution* operation checks wether new answers from the table space are available for consumption. When no new answers are available, the consumer node is *suspended* and execution proceeds using a specific strategy. Given the last consumed answer, we determine the unconsumed answer set R ($R \subseteq S$) and fetch the element $r \in R$, which is the first element from the set R. The last consumed answer can be seen as a *continuation* that is stored in each consumer node and is used to determine the next available answer.
- The *completion* operation determines if a tabled subgoal is *completely evaluated.* Only leader nodes can complete themselves and younger generator nodes. Once a subgoal is completed the set of answers S is closed and no more answers are accepted; future subgoal calls can use the set S without the need to suspend.

2.2.2 Tabling Instructions

Tabling engines extend the WAM instruction set with *tabling instructions* to support the four main tabling operations. Usually, each tabled predicate is compiled by using variants of the following instructions: `table_try`, `table_retry` and `table_trust`. These instructions are very similar to `try_me`, `retry_me` and `trust_me`, which the WAM natively implements.

For tabled predicates with multiple clauses, `table_try` is used on the first clause, `table_retry` for the middle clauses and `table_trust` for the last clause. Predicates with a single clause use a special instruction: `table_try_single`.

```
path/2_1:
  table_try_me path/2_2
  % WAM code for path(X,Z) :- edge(X,Y), path(Y,Z).
  new_answer
path/2_2:
  table_trust_me
  % WAM code for path(X,Z) :- edge(X,Z).
  new_answer
```

Figure 2.7: Compiled code for the tabled predicate `path/2`.

The `table_try` instruction implements the tabled subgoal call operation. If the subgoal is called for the first time, the data structures associated with this subgoal and a new generator node are created. Execution then proceeds by executing each predicate clause. This is performed by the `table_retry` instruction, which alters the next instruction of the generator choice point to the next clause of the predicate. On the other hand, if the subgoal is repeated, a consumer choice point is allocated and set to execute the `answer_resolution` instruction, which consumes answers and implements the answer resolution operation.

The instruction `table_trust` is executed by generator nodes and sets the next instruction to execute upon backtracking to `completion`, which runs the completion operation.

Finally, at the end of each clause, the instruction `new_answer` is appended to implement the new answer operation. This instruction has access to the arguments of a tabled subgoal call, thus by dereferencing them it obtains the corresponding answer, which can be added into the table space.

2.2.3 Scheduling Strategies

During evaluation of the previous example it is very clear that at several points we can choose between different *scheduling* strategies: continue forward execution, backtrack to interior nodes, return answers to consumer nodes, or perform completion. Depending on how and when the return of answers is scheduled, different strategies and searches can be formulated. It is also well known that using different strategies can lead to tremendous effect on performance as some predicates are better suited to specific strategies. The most popular scheduling strategies are *batched scheduling* and *local scheduling* [FSW96].

Batched scheduling reduces the need to suspend and move around the search tree by batching the return of answers. When the engine generates answers, while evaluating a particular goal, the answers are added to the table and the subgoal continues its normal evaluation until it resolves all available program clauses. Only then the answers are consumed by consumer nodes [FSW96]. In some cases, this results in creating dependencies to older subgoals, therefore enlarging the current SCC and delaying completion to older generator nodes. When backtracking, three situations may arise:

- if backtracking to a generator or interior node, try the next available clause.
- if backtracking to a consumer node, consume new answers.
- if no more clauses are left to try or no more unconsumed answers are available, two new situations may arise:
  - if the node is a leader node, attempt completion.
  - if not, backtrack to a previous branch.

Note that batched scheduling was the strategy used in the evaluation of the example in Figure 2.6.

For some problems, local scheduling is better suited because it tries to evaluate a single exact SCC at a time, preserving the dynamic SCC ordering during the evaluation. In other words, in a local evaluation, answers are returned to consuming nodes outside of an SCC only after that SCC is completely evaluated [FSW96]. It differs from batched scheduling in that once the answers are found, they are added to the table space, but execution *fails*. Because this strategy tries to complete sooner rather than later, we can expect less dependencies between subgoals.

2.2.4 Variant Tabling

When the subgoal `path(X,Z)` is called, a check for the presence of this subgoal in the table space is done first. In the example in Figure 2.6, this was done by checking wether a *variant* of the new goal already exists in the table. We say that two terms t_1 and t_2 are variants of each other if they are identical up to renaming of their variables.

For example, `path(X,Z)` is variant of the subgoal `path(X,Y)`, as they represent the same subgoal if we try to rename their variables to a standardized format. One format was proposed by Bachmair *et al* [BCR93]. Formally, we have a set V of variables present in a term and a function $renameVar$, such that the first term variable $v0 \in V$ results in $renameVar(v0) = VAR0$ and the following distinct variables are named incrementally $(VAR1, VAR2, ...)$. Using this mechanism, `path(X,Z)` and `path(X,Y)` both result in `path(VAR0,VAR1)`. The resulting standardized subgoal is then checked against the table space to verify if it is a repeated subgoal call.

The variant approach is widely used in tabling systems, but other approaches do exist. One approach named *call by subsumption* works by checking wether the new goal is subsumed by another goal in the table space. In other words, we verify if there is a more general subgoal than the one being called. For example `path(b,Z)` is subsumed by the subgoal `path(X,Z)`.

In this approach, instead of creating a new generator node at step 3, we create a consumer node that would consume answers stored in the subgoal `path(X,Z)`. For correct results, it should be clear that the answers used from the table space must unify with `path(b,Z)`. Like variant tabling, those new consumer nodes do not expand by using the program clauses, hence the search tree for this new method will be greatly reduced.

Variant and subsumption-based tabling define the *call similarity* property of the tabling engine. In a nutshell, when a subgoal A is declared to be *similar* to subgoal B, we say that A consumes from B (A is a consumer) and B generates its own answers.

2.2.5 Yap and XSB

Yap [SDRA] and XSB [SWS$^+$] are two well known Prolog systems that implement tabling.

The YAP Prolog system is a high-performance Prolog compiler developed at the University of Porto. It is one of the fastest available Prolog systems and implements a wide range of functionalities: stream I/O, sockets, modules, exceptions, debugging, a C-interface, dynamic code, internal database, DCGs, saved states, co-routining, arrays, threads and tabling. It is based on the WAM and follows the Edinburgh tradition. Most of the ISO-Prolog standard is implemented.

Tabling in Yap is implemented through the YapTab sub-system [RSS05b], a suspension based tabling engine supporting evaluation of definite programs. YapTab follows the seminal SLG-WAM (Linear resolution with Selection function for General logic programs in WAM) design from XSB Prolog [SSW96, SS98], but it innovates by proposing a new fix-point check algorithm, and by considering that the control of fix-point detection should be performed at the level of the data structures corresponding to suspended sub-computations. YapTab was originally designed to achieve good results in sequential tabling, but could be extended with the OPTYap engine, for parallel execution [RSS05b]. Other innovations in YapTab include: support for a dynamic combination of batched and local scheduling [RSS05a] and efficient handling of *incomplete tables* [Roc06a]. Incomplete tables are created when the current computation is pruned from the execution stacks, keeping the pruned subgoals from retrieving the complete answer set. Currently, only call by variant checking is supported.

XSB is a research-oriented logic programming system for Unix and Windows based systems. In addition to providing all the functionality of the Prolog language, XSB contains several features not usually found in logic programming systems, namely, evaluation according to the Well-Founded Semantics (tabling with negation) [GRS91] through the use of a delaying-based tabling engine, the SLG-WAM. Other features of XSB include: a fully threaded engine, constraint handling for tabled programs on a engine level, a variety of indexing techniques, interfaces to other languages, various compiler directives like `auto_table`, which does static analysis to decide which predicates to table, etc. [SWS$^+$].

SLG-WAM supports both tabling by variant checking and by subsumption checking. Tabling by call-subsumption was initially implemented by a technique called *Dynamic Threaded Sequential Automata* [RRR96] and is currently implemented using *Time Stamped Tries* [JRRR99].

In terms of design, both YapTab and SLG-WAM introduce the following extensions to the traditional WAM machine: the table space; a new set of registers, the *freeze registers*, one per stack (local stack, heap and trail); an extension of the standard trail, called the *forward trail*; and the tabling operations: *tabled subgoal call*, *new answer*, *answer resolution*, and *completion*.

The set of freeze registers says where stacks are frozen and protect the space belonging to suspended computations until the completion of the appropriate SCC takes place. They need to be adjusted in two different situations: when a computation suspends, increasing the portion of frozen stacks; and when a completion takes place, releasing part of space previously frozen.

The forward trail is used to restore all the variable bindings to their state at the time the computation was suspended. Thus, the WAM trail is extended with parent trail entry pointers to create this new trail. Also, a new register is created, the **TR_FZ** trail freeze register.

The differences between SLG-WAM and YapTab reside in the data structures and algorithms used to control the process of leader detection and the scheduling of unconsumed answers. These differences are described next in more detail.

2.2.5.1 SLG-WAM

The SLG-WAM considers that evaluation control should be done at the level of the data structures corresponding to first calls to tabled subgoals, and does so by associating *completion frames* to generator nodes [SS98].

The *completion stack* maintains, for each subgoal S, a representation of the deepest subgoal S_{dep} upon which S or any subgoal on top of S may depend.

When S and all subgoals on top of S have exhausted all program and answer clause resolution, S is checked for completion. If S depends on no subgoals deeper than itself, S and all subgoals on top of S are completely evaluated. Otherwise, if S_{dep} is deeper in the completion stack than S, S may depend upon subgoals that appear below it in the completion stack, and cannot be completed [SS98].

A one-to-one correspondence exist between completion stack frames and generator nodes, as the completion stack frame is pushed onto the stack when a new tabled subgoal is called. A completion frame is popped off when a subgoal is completed. Also, each subgoal frame contains a pointer to completion frame.

Consumer and generator choice points are extended to support the suspend and resume mechanism. The generator choice contains the following extra data: an explicit pointer of the failure continuation to take upon backtracking out of the choice point; a cell that records the value of a new global register, called the **RS** (*root subgoal register*) register, which points to the root subgoal of the node currently under execution; a pointer to the subgoal frame; a set of freeze registers, so that the stored values can be restored later on; and an area called the *substitution factor*, the set of free variables which exist in the terms in the argument registers.

The consumer choice point is extended with: a copy of the **RS** register; a pointer of the failure continuation to take upon backtracking; a substitution factor; the last consumed answer continuation; and a pointer to chain together all consumer choice points of the same subgoal.

2.2.5.2 YapTab

In YapTab, it is considered that the control of leader detection and scheduling of unconsumed answers should be performed through the data structures corresponding to repeated calls to tabled subgoals, and it associates a new data structure, the *dependency frame*, to consumer nodes [RSS00]. Each consumer choice point thus contains a field that points to the respective dependency frame.

Dependency frames are used to check for completion points and to move across the consumer nodes with unconsumed answers, thus they are linked together, forming the *dependency space*.

Generator choice points are WAM choice points extended with the substitution factor area, a pointer to the subgoal frame and, a pointer to a dependency frame. This pointer is only used when local scheduling is employed. A generator node for local scheduling only exports its answers to the calling environment when all clauses for the subgoal have been exhausted, hence it must act like a consumer.

In SLG-WAM, if we want to release space previously frozen and restore the freeze registers, we use the stack values stored in the generator choice point to perform completion. In YapTab, as the freeze registers are not saved there, we use the top stack values kept in the youngest consumer choice point younger than the current completion point.

YapTab reduces the size of consumer choice points by using dependency frames. Each frame contains the following fields: **last_answer** field that stores the last answer consumed by the choice point; the **consumer_cp** field that points to the consumer choice point; the **leader** field which points to the leader node at creation time; and the **back_leader** field that changes during evaluation, pointing to the leader node where we performed the last unsuccessful completion operation. A new global register, called **TOP_DF**, always points to the youngest dependency frame.

2.3 Chapter Summary

In this chapter we introduced logic programming and Prolog. We discussed the shortcomings of the SLD resolution and then we presented tabling, which is a technique that solves these problems.

Next, we discussed how tabled programs are evaluated, by introducing the concepts of generator, consumer, scheduling strategies, such as batched scheduling and local scheduling, and the call similarity property in variant and subsumption-based tabling engines. We also discussed two of the most widely known Prolog systems that implement tabling, Yap and XSB Prolog.

In the next chapter we will throughly discuss and analyze how the table space is organized to efficiently implement variant-based tabling and then subsumption-based tabling.

Chapter 3

Table Space Organization

In this chapter, the table space organization for both variant and subsumption tabling engines are described. First, the variant table space for both SLG-WAM and YapTab is explained. Because they share a lot of similarities, the description covers the common ground between the two systems. Next, we explore two well-known mechanisms that modify the previous table space for call by subsumption. Those two mechanisms have already been implemented in XSB [RRR96, JRRR99].

3.1 Table Space

Implementing a tabling engine on a Prolog system involves the design of compact and time efficient data structures to organize the table space. The table space is heavily used throughout the evaluation process in various operations:

- to lookup if a subgoal is in the table, and if not insert it;
- to verify whether a newly found answer is already in the table, and if not insert it;
- to retrieve answers for consumer nodes.

3.1.1 Tries

Clearly, the success of tabling is highly dependent on the data structures used. Both Yap [RSS00] and XSB [RRS^{+}95] use a trie-based approach.

Tries were initially proposed to index dictionaries [Fre62] and have since been generalized to index recursive data structures such as terms. The essential idea underlying a trie is to partition a set T of terms based upon their structure, in such a way that common term prefixes are represented only once.

A trie is a tree-structured automaton with the root as the start state and each leaf state associated with a term in T. Each state specifies the position to be inspected in the input term on reaching that state. The outgoing transitions specify the function symbols expected at that position. A transition is taken if the current symbol in the input term matches the symbol of the transition. If we recursively reach a leaf state we say that the input term *matches* the term represented by the leaf state. A complete path, from the root to a leaf, corresponds to a pre-order traversal of the matching term. If no transition can be taken, the lookup operation fails. On the other hand, for an insert operation we add a new outgoing transition for the current input symbol and a new node, which is linked to this transition. To complete the insert operation, we consume the rest of the input term, until a leaf node is created that represents the newly inserted term.

Given the nature of tries, the following conclusions can be made:

- the lookup/insert operation can be done in a single pass through a term. If the lookup fails, it is possible to complete an insert operation using the last lookup state;
- the efficiency and memory consumption of a particular trie depends on the percentage of terms that have common prefixes.

When creating transitions for variables, we use the format outlined by Bachmair *et al.* [BCR93], described in Section 2.2.4.

Figure 3.1 shows a trie with three terms. First, in (a) the trie is represented by a *root node* and has no terms. Next, in (b) the term `t(X,a)` is inserted and three nodes are created that represent each part of the term. In (c) a new term, `u(X,b,Y)` is inserted. This new term differs from the first one and a new distinct branch is created. Finally, in (d), the term `t(Y,1)` is inserted and only a new node needs to be created as this term shares two prefix nodes with `t(X,a)`.

Yap and XSB use two levels of tries to implement the table space (see Figure 3.2):

- The first level (subgoal trie) stores subgoal calls for each predicate;

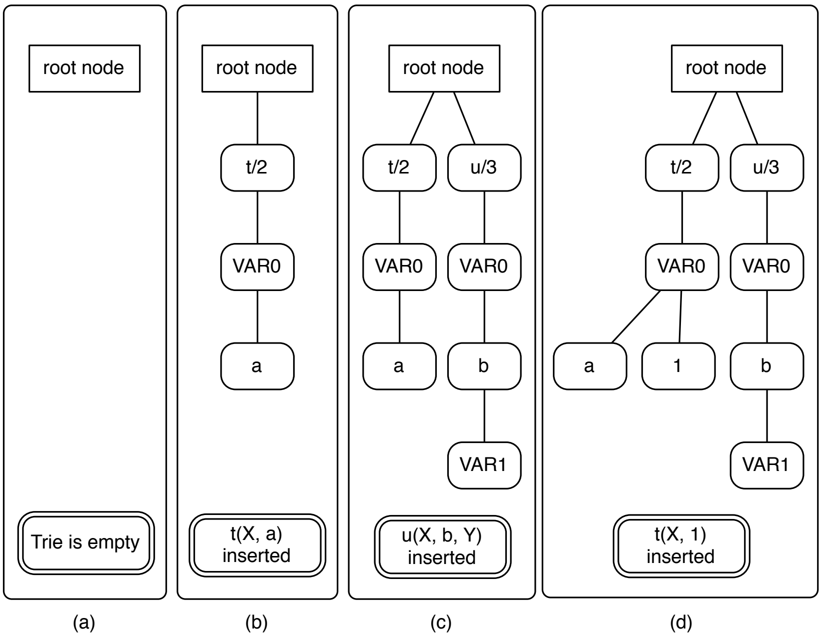


Figure 3.1: Using tries to represent terms.

- The second level (answer trie) stores answers for a specific subgoal.

For both levels, each trie node usually contains four fields. The first field represents the **symbol** (or **atom**) of the transition. The second points to the first descendant transition (called the **child** node) and the third stores a pointer to the **parent** node. The fourth field points to a **sibling** node, if any.

When the chain of sibling nodes gets too big, an hashing scheme is dynamically employed to provide direct access to nodes, optimizing the search of transitions.

Each tabled predicate contains a *table entry* that points to a *subgoal trie*. Each different call to a tabled predicate corresponds to a unique path through the subgoal trie. Notice that for the subgoal trie only the subgoal arguments are stored. The leaf points to a data structure called *subgoal frame*. The subgoal frame stores information about the subgoal, namely an entry point to its *answer trie*.

The answer trie stores answers to the subgoal. When inserting answers only substitutions for the variables in the call are stored. This optimization is called *substitution factoring* [RRS+95].

Figure 3.2 shows the table space organization after evaluating the query goal `path(X,Z)` for the example in Figure 2.6. The figure shows how the subgoals `path(X,Z)`, `path(b,Z)`

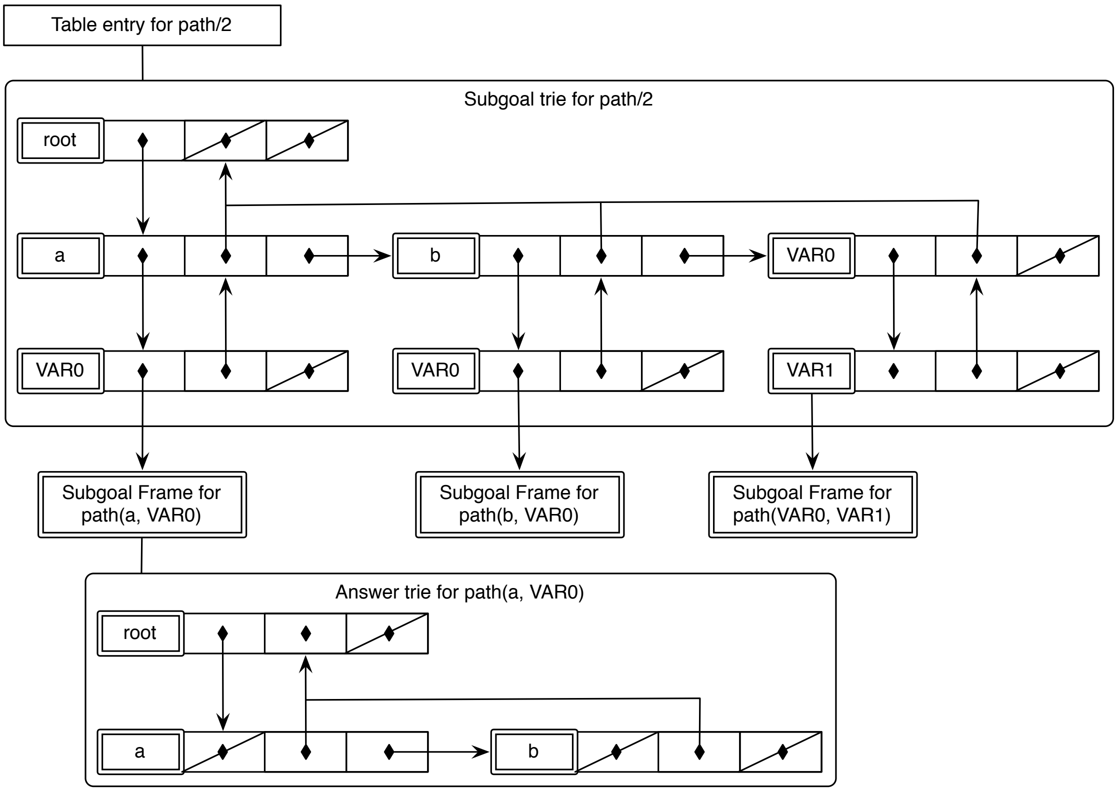


Table entry for path/2
Subgoal trie for path/2
root
a
b
VAR0
VAR0
VAR0
VAR1
Subgoal Frame for path(a, VAR0)
Subgoal Frame for path(b, VAR0)
Subgoal Frame for path(VAR0, VAR1)
Answer trie for path(a, VAR0)
root
a
b

Figure 3.2: Organizing the table space with tries for variant tabling.

and `path(a,Z)` are stored in the `path/2` subgoal trie and how the answer trie for `path(a,Z)` is represented. Notice that by using substitution factoring for the answer trie, only the substitutions {`VAR0 = b`} and {`VAR0 = a`} need to be stored.

3.1.2 Subgoal Frames

A subgoal frame contains general information about the state of a tabled subgoal. To access answers, this frame contains a pointer to the root of the answer trie. A chain of answers used by consumers is also kept in the form of head and tail pointers. In XSB, a *answer return list* is built and the consumer has a pointer to the node of the list representing the last consumed answer. In Yap, answers are chained using the **child** pointer of the leaf answer nodes and consumers only keep the last consumed answer leaf. The last consumed answer pointer is an *answer continuation*. When a consumer needs to verify or consume the next available answer, it uses the continuation to retrieve the next answer, following the chain of answers. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

To simplify memory management, both systems link subgoal frames by storing **next** and **previous** pointers in each subgoal frame, forming a double linked list. The evaluation state of the subgoal is also stored. For example, Yap has the following states: *ready*, *evaluating*, *complete* and *incomplete*.

3.2 Tabling by Call Subsumption

Although variant based tabling has proven to be greatly beneficial in solving some shortcomings of the SLD resolution, other approaches are possible. Tabling by call subsumption aims to reuse answer computations by sharing answers from *more general* goals [JRRR99].

When a subgoal is first called, a variant engine will lookup in the table space for a variant subgoal, i.e., one that is identical by renaming the variables. If such a subgoal already exists on the subgoal trie, a new consumer is created and the available answers from the variant subgoal are pushed for consumption.

Although a variant check is a light-weight operation computationally, tabling engines using such checks can end up computing answers through program clause resolution, which takes time and space, when they could retrieve answers from a subgoal that *subsumes* the new call. By other words, a more specific subgoal could consume answers from a general subgoal, which contains the full set of answers for the specific subgoal among the complete set.

Formally, if two subgoals G and G' exist, such that S and S' are the respective answer sets and G' subsumes G, then we can conclude that $S \subseteq S'$.

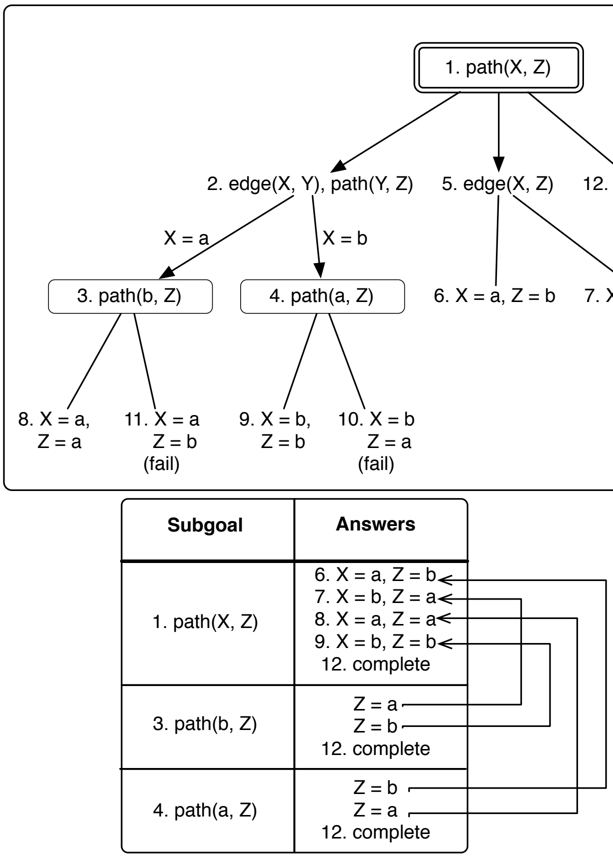
The effects of using subsumptive checks are greater reuse of computed answers and reduced program clause resolution, yielding superior time performance and memory usage. In terms of memory usage, improvements can be made since fewer calls and their associated answer sets need to be preserved [JRRR99]. However, implementing call subsumption poses various challenges:

(a) How to efficiently check for subsuming subgoals in the subgoal trie;

(b) How to design new mechanisms to represent answers supporting fast retrieval of subsets that are only related to a subsumed call;

(c) How to support incremental retrieving of the answer subset. Note that, during evaluation, it may be not possible for the subsuming call to contain all answers, as the process of generating answers is incremental.

For illustration purposes, in Figure 3.3, we describe the evaluation of `path(X,Z)` using the program presented in Figure 2.4 and compare it against the variant approach in Figure 2.6.



Subgoal	Answers
1. path(X, Z)	6. X = a, Z = b 7. X = b, Z = a 8. X = a, Z = a 9. X = b, Z = b 12. complete

3. path(b, Z)	Z = a
	Z = b
	12. complete

4. path(a, Z)	Z = b
	Z = a
	12. complete

Figure 3.3: Tabling `path(X,Z)` using call by subsumption.

At step 1 the subgoal `path(X,Z)` is called and a new generator node is created as there is no existing subgoals in the table space that could be variant or subsuming. Being a generator node, it is evaluated using the program clauses (step 2). The first solution for the `edge(X,Y)` predicate is evaluated using Prolog's standard rules and yields a subgoal call to `path(b,Z)` (step 3).

In variant tabling the engine would search for a variant subgoal in the table space, thus failing and creating a new generator node, meanwhile expanding the execution tree by means of program clause resolution. In subsumptive tabling, the engine searches for a subsuming call, and finds `path(X,Z)`. As this subgoal is more general than `path(b,Z)` it will generate all the answers needed for the subsumed goal. A special type of consumer called a *subsumptive consumer* is thus created. This consumer knows the subsuming subgoal and can retrieve the answers that unify with the subsumed call.

As `path(X,Z)` has no answers, no answers can be consumed by `path(b,Z)`, and thus the consumer node is suspended and execution backtracks to node 2. The second clause of `edge/2` is tried and a new tabled subgoal is called: `path(a,Z)` (step 4). Like `path(b,Z)`, this subgoal is subsumed by `path(X,Z)`, hence a new subsumptive consumer node is created. Execution suspends once again because no answers are available and the engine goes back to node 1 to try the second `path/2` clause (step 5).

Answers for `path(X,Z)` are found in steps 6, {`X = a, Z = b`}, and 7, {`X = b, Z = a`}. They are stored in the answer set for this subgoal. Execution thus backtracks to node 1 and completion is attempted. Node 1 verifies that node 3 and 4 have unconsumed answers and starts by resuming computation at node 3.

Node 3 being a subsumptive consumer looks up in the `path(X,Z)` answer set for answers specific to `path(b,Z)`, finding one: {`Z = a`}. At this point, the consumer marks the answer for later retrieval, so that answers can be immediately consumed when a variant subgoal is called. Like variant tabling, the consumer keeps a last consumed answer continuation to know which answers have already been consumed. Variable bindings for this answer are propagated and a new answer to `path(X,Z)` is found: {`X = a, Z = a`} (step 8). Node 3 tries to consume a new answer, but no new answers are available.

Execution suspends node 3 and resumes computation at node 4. Here `path(a,Z)` begins to consume answers from `path(X,Z)` using the subsumption mechanism. The first answer in `path(X,Z)`, {`X = a, Z = b`}, unifies with the subsumed goal and is consumed. By variable propagation, a new answer for `path(X,Z)` is also generated, {`X = b, Z = b`} (step 9). Node 3 then tries to consume a new answer and finds {`X = a, Z = a`}. This new answer is added to the `path(a,Z)` table space. As the new variable bindings are propagated, a new answer is also generated for top subgoal, but the answer {`X = b, Z = a`} is repeated (step 10), hence it is not inserted into the table space.

Execution now suspends node 4 and resumes the the leader node, `path(X,Z)`, that will once again attempt completion. By using the last consumed answer continuation, it verifies that node 3 has new unconsumed answers. Execution thus resumes again at node 3.

Node 3 inspects `path(X,Z)` answer set and consumes the next matching answer, {`X = b, Z = b`}. A new answer for `path(b,Z)` is generated and variable bindings are propagated to the leader node (step 11). However, the new answer is already stored in the table space and it is discarded.

Once again, evaluation returns to the leader node and completion is performed (step 12) as each consumer has exhausted the available answers.

This evaluation example, when compared to variant tabling, shows a smaller execution tree with less program clauses expanded. The subsumptive computation also took less steps to complete and a greater reuse of answers was done between `path(X,Z)`, `path(b,Z)` and `path(a,Z)`.

Like variant tabling, the same four main operations are used when evaluating subsumptive subgoals. These operations are very similar, with a few differences. Operations *new answer* and *completion* remain the same, while the *tabled subgoal call* and *answer resolution* operations work slightly differently:

- The *tabled subgoal call* operation represents a call to a tabled subgoal. When a subgoal C is called, a search for a subsuming subgoal C' is done. If such subgoal is found, C will be resolved using *answer clause resolution*, thus consuming answers from C'. If no subgoal C' is found, then a new entry, representing C, is inserted into the table space and C will be evaluated using program clause resolution.

- The *answer resolution* operation checks wether new answers from the table space are available for consumption. Each subsumptive consumer node uses an answer continuation that represents the set of remaining answers to be consumed. Given this answer continuation, we inspect the answer trie from the subsuming subgoal and retrieve the next answer that matches with the subsumed goal. Once the answer is retrieved and loaded, the answer continuation is updated to reflect the new answer consumed.

We next describe two known techniques that implement subsumptive tabling. These two approaches were both implemented in XSB.

3.2.1 Dynamic Threaded Sequential Automata

Dynamic Threaded Sequential Automata (DTSA) is a data structure that provides good indexing and incremental returning of a subset of answers from a subsuming call to a subsumed call [RRR96].

This structure orders answers as they are generated, hence it can easily mark which answers were already retrieved, enabling efficient retrieval of the remaining answers.

A DTSA is based on a *Sequential Factoring Automaton* (SFA) [DRR+95] but has a few more features for dealing with fast indexing and incremental retrieving of answers.

3.2.1.1 Sequential Factoring Automaton

A SFA solves the problem of retrieving an ordered subset of answers, starting from the oldest to the newest answer. It is an ordered tree-structure automaton and is very similar to a trie. It begins with a root as the start state and has edges as transitions that represent unifications. Every leaf represents a distinct term and the transitions on the path from the root to a leaf represent the operations necessary to unify the goal with the term at the leaf.

Apart from ordering, SFAs differ from tries in that transitions from a state may not be unique and each transition in a trie denotes match operations, not unify operations.

To insert a new term into a SFA, we must start by inserting symbols from the answer term into the root state. In each state we verify if the *last transition* s matches the current term symbol t. If t matches s we take the transition and advance to the next state; if not, a new transition for t is created and we advance to the newly created state. When all term symbols t are exhausted, the current state is marked as a leaf, representing a new answer. Figure 3.4 represents a SFA for the subgoal `p(X,Y,Z)` and illustrates insert operations.

When a new subgoal G subsumed by the subgoal G' is first called, we must determine the *answer template* from G' to the sub-terms of G, because only variable substitutions are inserted into a SFA. The answer template is usually stored under the choice point.

For example, if G' is `p(X,Y,Z)` and G is `p(a,a,V)`, the variable assignments are {`X = a, Y = a, Z = VAR0`}. During unification operations, we must unify the first SFA symbol to a, then to a again, and finally with `VAR0`. Once a variable is *bound*, subsequent unify operations must unify with the bounded sub-term.

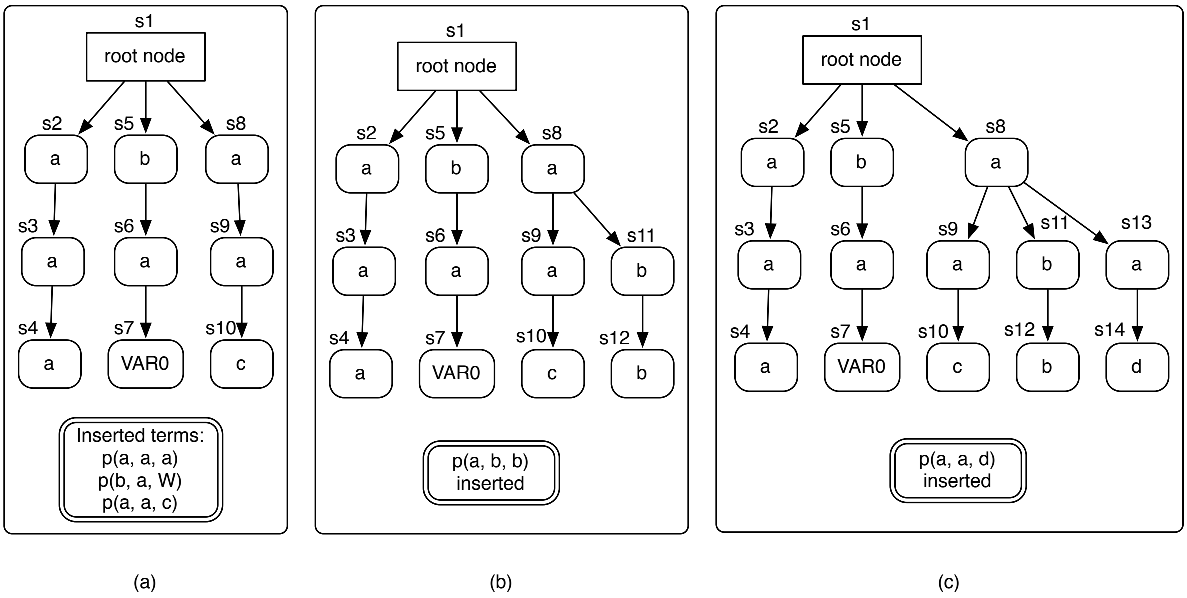


Figure 3.4: Inserting answer terms in a SFA.

The unification process starts in the root state with an empty *continuation stack*. When a state s is reached, the leftmost transition that unifies with the current G assignment is chosen, this is called the *applicable transition*. Before moving to the next state, we select the next applicable transition and push it on the continuation stack. If no applicable transitions are available at state s or if an answer was found, we pop a transition from the stack and use it to search for more answers. Once no more transitions can be taken and the stack is empty, the search process ends.

Using the SFA in Figure 3.4 and the subgoal G, `p(a,a,V)`, the search mechanism starts at the root state and is ready to retrieve all answers that unify with G. The first applicable transition is $s1 \rightarrow s2$ because it unifies with the first symbol: a. The transition $s1 \rightarrow s5$ can not be pushed into the continuation stack because it does not unify, but $s1 \rightarrow s8$ does. In state $s2$ only one transition is available, thus nothing is pushed into the stack. Transition $s2 \rightarrow s3$ unifies with symbol a and we move to state $s3$. Here, the variable `VAR0` (that represents `V`) can unify with a, thus we can get to state $s4$, arriving at a leaf state and a new answer, {`V = a`}.

Next, the process must use the continuation stack to retrieve more answers. Transition $s1 \rightarrow s8$ is popped from the stack and we arrive at state $s8$ with 3 available transitions. Using the previous rules, a new answer is retrieved, {`V = c`} and the continuation stack contains the transition $s8 \rightarrow s13$.

Finally, once states $s13$ and $s14$ are visited, the process arrives at a leaf state and a new answer, {`V = d`}, is retrieved. The process finishes and every answer that is specific to `p(a,a,V)` is found.

3.2.1.2 Threaded Sequential Automata

A *Threaded Sequential Automata* extends the SFA with a concept called *equivalent states*. One state S_1 is equivalent to state S_2 if when S_1 is taken, S_2 is also guaranteed to be visited. For example, in Figure 3.4, whenever states $s2$ and $s3$ are visited, states $s8$ and $s9$ are also guaranteed to be visited, as they denote the same path.

A SFA is converted to a TSA by adding *equivalence links* between equivalent states. The SFA in Figure 3.4 was transformed into a TSA in Figure 3.5.

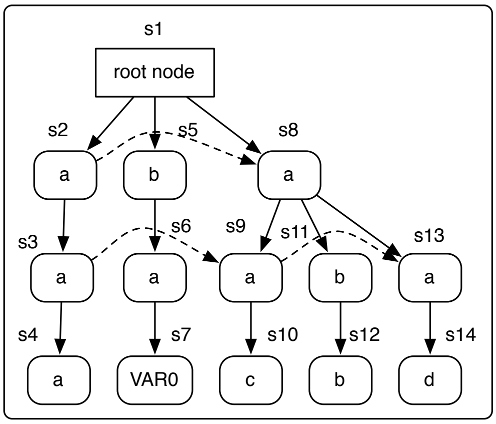


Figure 3.5: SFA transformed into a TSA.

In the TSA, the concept of applicable transitions is changed. Now it is also possible to push equivalence links into the continuation stack as if they were normal transitions. So, if we were at state $s3$ we use the transition to $s9$ and then to $s13$ instead of going from the start state.

Although equivalence links provide an efficient indexing mechanism, they must be used with care. If not, some situations arise where following equivalence links lead to repeated answers and answers in the incorrect order [RRR96]. The selection of transitions must consider only *safe transitions*, which reach answers that cannot be reached through the pending transitions on the stack. So, if the process is at state $s2$ and the transition $s1$ to $s8$ is already on the stack, we can not use the equivalence link $s2 \rightarrow s8$, as the transition $s1 \rightarrow s8$ already covers the same branch.

If we followed only safe transitions, no equivalence links would be used, hence we must check if the usual next applicable transition covers some answers that can not be reached from using the equivalence links on the next branch to explore, thus favoring equivalence links. Notice that at state $s2$ we would use the equivalence link $s2 \rightarrow s8$, instead of the transition $s1 \rightarrow s8$, as $s2 \rightarrow s8$ will cover the same answers.

3.2.1.3 Dynamic Threaded Sequential Automata

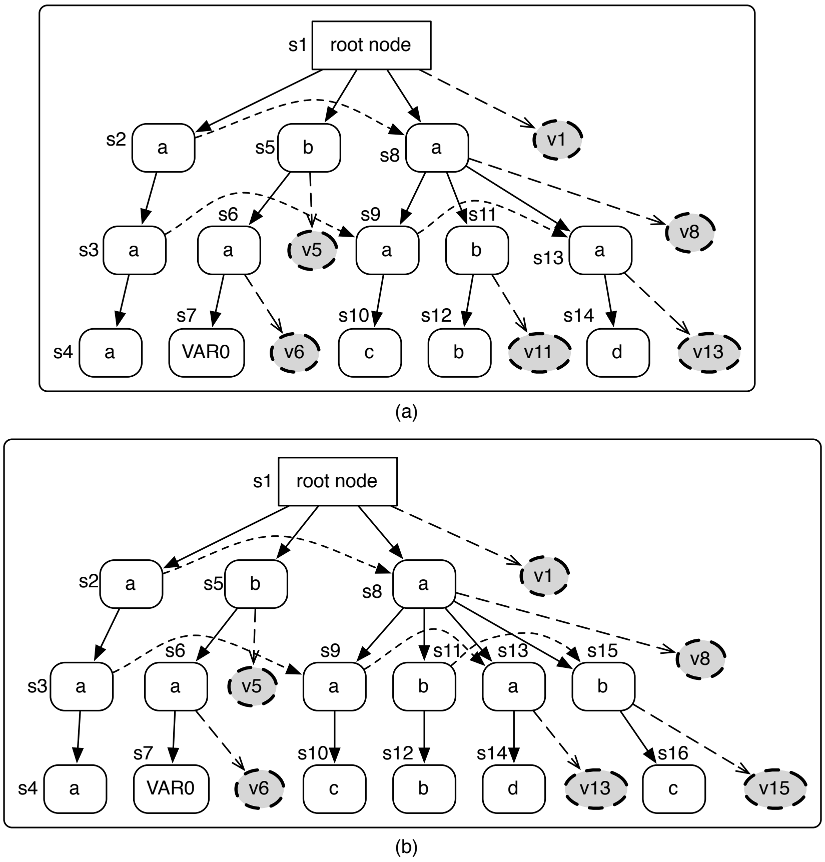
The previously described mechanisms only work if we have the complete set of answers for the subsuming subgoal. A new mechanism that deals with incomplete answer sets must be devised, so that after all current answers are retrieved, the continuation stack can be used to retrieve newly inserted answers.

The Dynamic Threaded Sequential Automata extends the TSA with special transitions in states were new transitions can be inserted or in states were no equivalence links exist. Figure 3.6 illustrates two DTSAs: (a) shows the converted TSA from Figure 3.5, and (b) the resulting DTSA after inserting the answer `p(a,b,c)`.

During answer retrieval, if no answer is found and the top of the continuation stack contains a transition to a special state, the process stops and the continuation stack is saved along the last state visited. Later on, when new answers must be retrieved, the last state is used to transform the stack to account for new states that were introduced during the insertion of new terms. If the new stack contains a valid transition, it can now be used as usual.

For example, retrieving answers to subgoal `p(a,X,c)` from the DTSA in Figure 3.6 (a) results in the answer `p(a,a,c)` and a continuation formed by the last visited state $s13$ and the stack containing (from bottom to top): $[s1 \rightarrow v1,\ s8 \rightarrow v8,\ s13 \rightarrow v13]$. This continuation stack contains the transitions from where it is possible that a relevant answer will be inserted, by using either normal transitions or equivalence links.

After the new answer `p(a,b,c)` is inserted into the DTSA in Figure 3.6 (b) and a consumer is resumed to consume new answers, we must check if new answers are available, hence the continuation stack must be transformed in order to convert the virtual transitions to instantiated transitions that may have been created after the last retrieval operation. This stack modification is done by using the last visited state $s13$, resulting in the following stack configuration: $[s1 \rightarrow v1,\ s8 \rightarrow s15]$. Now the answer `p(a,b,c)` can be easily retrieved using the transition $s8 \rightarrow s15$. Notice that



(a)

(b)

Figure 3.6: DTSA before and after inserting answer `p(a,b,c)`.

the transition $s1 \rightarrow v1$ remains unmodified as new transitions can be created from $s1$.

3.2.1.4 Table Space Organization

This new DTSA mechanism was implemented in XSB by extending the variant engine [RRR96]. First, each subgoal frame now contains both an answer trie and a DTSA. Answer tries are used to check for duplicate answers and the DTSA is created lazily, when a subsumed subgoal is first called.

Each subsumed subgoal keeps an answer return list that is built using the DTSA retrieval algorithm. This answer list is used when variant goals of the subsumed goal are called, thus instead of using the DTSA, answers are retrieved directly by traversing the linked list. When a subgoal is marked as complete, the DTSA is deleted and the answer trie is converted into WAM instructions, a feature called *compiled* *trie code* [RRS+99]. Answers for subsumed goals are then retrieved using the trie instructions through unification and backtracking.

3.2.2 Time Stamped Tries

Time Stamped Tries (TST) is another mechanism that was implemented in XSB to support tabling by call subsumption [JRRR99].

TST is a relatively simple technique based around the idea of augmenting a trie with information about the relative time its terms were inserted. The time of insertion of each term is called its *time stamp* and is represented by a positive integer. The time stamps are then used for incremental answer retrieval.

For each node in a trie, we extend it by including a time stamp. Along the augmented trie, the maximum time stamp T is also stored, thus allowing the insert mechanism to know the next time stamp to use for new trie paths. An example TST for the subgoal `p(X,Y,Z)` is represented in Figure 3.7. By looking at the leaf nodes, the order of answer insertion can be readily known: `p(a,a,a)`, `p(b,a,VAR0)`, `p(a,a,c)`, `p(a,b,b)` and then `p(a,a,d)`.

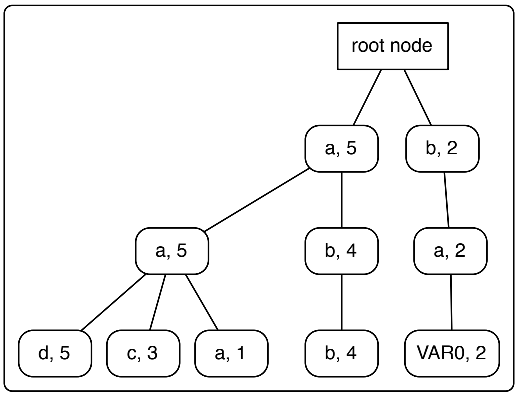


Figure 3.7: Time Stamped Trie for subgoal `p(X,Y,Z)`.

3.2.2.1 New Answers

The process of inserting a new answer into a TST starts by traversing matching nodes as long the stored symbols match the new answer. If the current symbol does not match, the process changes from search to insert mode and new nodes are inserted to represent a new trie path. Once the leaf node is created, each node from leaf to root is traversed and its time stamp is updated to $T + 1$.

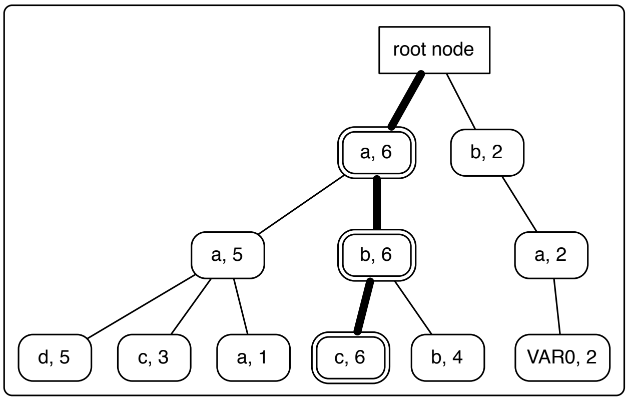


Figure 3.8: Time Stamped Trie from Figure 3.7 after inserting the answer `p(a,b,c)`.

Figure 3.8 shows the TST from Figure 3.7 after a new answer `p(a,b,c)` was inserted. Note that each node from the answer leaf node to root node was updated with time stamp 6. Apart from the time stamps, search and insertion in TSTs work exactly the same as standard tries.

3.2.2.2 Retrieving Answers

Retrieving all answers that are specific to a subsumed subgoal G from a TST with answers from subgoal G' works by navigating the TST and unifying the answers against the answer template. If we wanted to retrieve answers to the subgoal `p(a,a,X)` from the TST in Figure 3.8, during the subgoal call we determine the answer template relative to `path(X,Y,Z)`: {`X = a, Y = a, Z = VAR0`}, and store it under the choice point. Then, the terms `[a, a, VAR0]` are unified with the trie symbols, thus finding answers that are specific to G.

Like tries, TSTs only store substitutions for variables, thus we must unify the first sub-term a, then a again and then the variable, which unifies with any symbol. During the unification process, if a variable appears multiple times, it must unify with any previous sub-term assignment.

Time stamps guide the answer unification process by filtering transitions to already explored branches, where answers were already retrieved, thus avoiding repeated answers.

The unification process that finds new answers by using a time stamp is separated from the process of unifying the retrieved answers with the subsumed subgoal. This *two-tier mechanism* is key to the space and time efficiency of the design of TSTs [JRRR99] and allows identification of all relevant answers that have been added since the last time a search operation was done by using the time stamp.

3.2.2.3 Table Space Organization

The table space in this technique extends the variant table space by using TSTs instead of answer tries for subsuming goals.

Each subsumed subgoal in a subgoal trie stores the last search time stamp t. The process of incrementally searching for new answers in a TST will use t and update it after the process completes. When a subsumed subgoal is first called t is set to 0, thus initially allowing the retrieval of all relevant answers.

The subgoal in the subgoal trie also stores an answer return list. Each time new answers are identified, they are appended to this linked list. The original subsumptive consumer and its variant subgoals will then consume answers from it. If no new answers can be retrieved from the list, the TST process is employed to identify more answers from the subsuming TST, inserting them into the list.

An hashed TST node maintains a time stamp index which stores all transitions in reverse order. It is not until a subsumed subgoal is first called that all time stamp related structures are created, thus allowing a more efficient use of space.

Like DTSAs, the TST indexing mechanism is only used during the evaluation of subsumed calls. For calls with completed tables, subsumed subgoals will use compiled trie instructions from the more general subgoal.

The main advantage of using TSTs instead of DTSAs is in terms of space complexity. In TSTs, the maximum table space used is at most twice that of the variant engine, because each node can have both the time stamp and a time stamp index node. For DTSAs, the space used is at least double, but in the worst case can be quadratic. DTSA is at advantage in terms of speed, because it supports identification of answers and unification in one step, thus answers can share some elementary unifications. In TSTs, identifying answers and doing answer unification is a two step process, thus it takes more time to construct all answers.

3.3 Chapter Summary

In this chapter we explored how the table space is efficiently designed to support the tabling operations in a variant or a subsumption-based engine. For the variant engine, we presented the YapTab's table space that is based on two levels of tries, the subgoal trie and the answer trie.

Next, we discussed the two most well-known table space organizations to implement call subsumption. The first technique is the Dynamic Threaded Sequential Automata (DTSA). This method creates a special data structure based on tries to efficiently collect the relevant answers for a subsumed subgoal. The second technique is the Time Stamped Trie (TST). It improves upon the previous technique by being simpler and having reduced space needs.

In the next chapter we will discuss the algorithms and data structures used in the TST technique, covering the algorithm used to discover subsuming subgoals and the algorithm to collect relevant answers. Finally, we will explain how the TST is integrated into the YapTab tabling engine to enable call by subsumption.

Chapter 4

Time Stamped Tries

In this chapter, we throughly describe the Time Stamped Tries approach to implement a subsumptive tabling engine. This mechanism was proposed by Ernie Johnson *et al.* [JRRR99] and is currently implemented in XSB. It is based on the idea of extending each answer trie node with *time stamp* information as a means to distinguish between new answers and old answers.

First, we start by describing the algorithms and data structures associated with the detection of subsuming goals. Next, we explain the data structures introduced in the answer tries to support subsumption, focusing on answer insertion and retrieval for subsumed subgoals. Finally, we describe the modifications made to the YapTab tabling engine in order to support subsumptive tabling, focusing on the tabling operations and on the table space.

4.1 Finding General Subgoals

The problem of finding on a subgoal trie C a subgoal G' that subsumes G is an important part of a subsumptive tabling engine, as it makes possible to identify a subsumptive relation between G' and G.

In the SLG-WAM, the search is performed by recursively backtracking through the subgoal trie C, trying to match the node symbols with subterms from G. The process stops once a leaf node is reached.

The matching process gives priority to match non-variable terms from G with an identical symbol from C. Alternatively, if the current trie symbol is a variable, for example VAR0, on its first occurrence, VAR0 is bound to the respective G sub-term and match succeeds. On the next occurrences, the current sub-term from G must be identical to the term bound to VAR0. Throughout the search process, bound variables are matched before new unbound variables.

Favoring non-variable terms, results in a mechanism that finds *minimally subsuming calls*. In particular, if there is a variant trie path of G called G'', G'' is found before any other subgoal. If no variant path exits, it is possible to speedup the process of inserting a variant path of G by recording the trie node where: (1) the first constant match failure occurred; or (2) an already seen G variable paired to a trie variable could not be paired to the same trie variable.

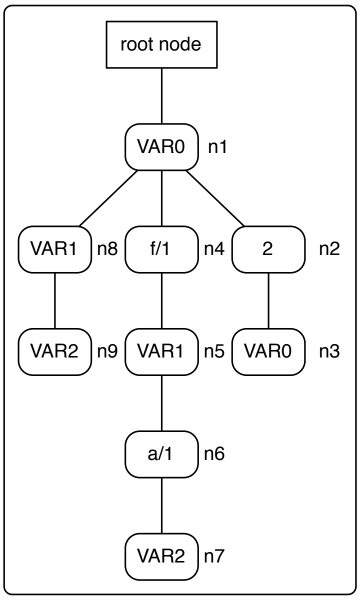


Figure 4.1: Subgoal trie for a tabled predicate p/3.

For illustration purposes, Figure 4.1 shows a subgoal trie for a tabled predicate p/3. The first subgoal called was p(X,2,X), followed by p(X,f(Y),a(Z)) and finally p(X,Y,Z).

If the subgoal p(X,2,X) is called again, the algorithm described above should find the variant subgoal represented by the leaf node $n3$. First, we unify the trie variable VAR0 with X (node $n1$), hence we must mark the variable X as *seen*, because if the same variable appears again it must unify with the same trie variable for a variant path to exist. Next, 2 easily unifies with trie node $n2$ and unification proceeds. In trie node $n3$, the current call term is X and we also have a variable in the trie node. As

X was already seen before, it must unify with VAR0. It does and thus a variant path is found. Please note that if the trie symbol at node $n3$ was VAR1 no variant path would exist, but the process could proceed and a subsuming subgoal would be found, as p(VAR0,2,VAR1) subsumes path(VAR0,2,VAR0). In this case, a variant path could be created by resuming the insert operation at node $n2$ to insert a VAR0 node.

For a more complex example, let the subgoal p(2,f(X),a(2)) now be called for the same subgoal trie. First, the algorithm searches for a trie node with the symbol 2, and as it does not find one, no variant path exists in this subgoal trie. Next, the algorithm tries to unify with bound variables, but as the process has just started, only unbound variables are found, and VAR0 (node $n1$) is unified with 2. The functor term f/1 is the next term on the subgoal and node $n4$ matches with it. The next term is X and it can unify with VAR1 (node $n5$). Note that if we failed at this point, the process would backtrack to node $n1$ and node $n8$ would be tried next, which would lead to a more general subgoal. Next, the functor term a/1 matches with trie node $n6$ and the process proceeds. The last term symbol 2 can match with node $n7$, as it is an unbound variable and the only node available. If the variable was bound, like VAR0 for instance, the process would check if the current term symbol unifies with the variable binding made before (VAR0=2) and it would also succeed. As node $n7$ is a leaf node, the process finishes with a subsumptive path found and the following variable bindings made: VAR0=2, VAR1=X, and VAR2=2.

Implementation wise, this algorithm uses the following data structures:

- *variable bindings vector*: saves bindings for each numbered trie variable. Starts with each position pointing to itself;
- *variable enumerator vector*: when a never seen term variable appears it must be bound to a position in this enumerator, ensuring that it can be recognized if the same variable appears a second time;
- *term stack*: stores the remaining terms to be unified against the trie symbols;
- *term log stack*: stores already matched terms taken from the term stack. Each frame contains the top index and the top element of the term stack during the creation of a frame;
- *trail stack*: stores bindings that were made during the process. It is used to untrail variables during backtracking;

- *call choice point stack*: used to restore the search process at a certain node to explore alternatives.

During execution, two matching methods are considered: the first tries to match exact trie symbols against the current term; the second method uses trie variables instead of exact symbols and is employed when the first method fails. When a trie node match succeeds, matching defaults to the first method, which means that exact matches are always tried before variables when a new *trie level* is reached and variables are usually attempted when backtracking. A trie level represents a set of nodes that are linked by **sibling** links or that are in the same hash table.

The process starts by pushing the G subgoal arguments, $X1, X2, ...Xn$ into the term stack, so that $X1$ is at the top and Xn at the bottom (Figure 4.2). Then, the algorithm proceeds in a modified depth-first manner, trying to match the exact nodes first, and then the variable trie nodes. The skeleton for this algorithm is presented in Figure 4.3.

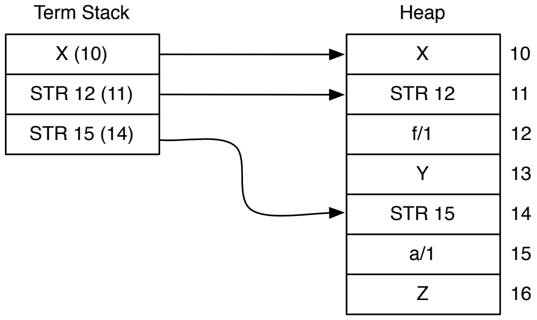


Figure 4.2: Initial term stack and heap representation for subgoal `p(X,f(Y),a(Z))`.

The basic idea of this algorithm is to match the current term from the term stack against a node in the current trie level and store the next alternative node on this level on the call choice point stack. If no node is found, we try alternatives on the choice point stack, else we use the matched node to descend into the trie. Two modes of matching exist: (1) **MATCH_EXACTLY**, which is used when a new trie level is reached for the first time in order to do exact matches; (2) **MATCH_TRIE_VARS**, used when the current trie node is reached by backtracking, forcing variable matching.

4.1.1 Call Choice Point Stack

The call choice point stack (Figure 4.4) contains alternative search paths to use if the process fails somewhere in the trie. Each stack frame can resume the search at a given

```
lookup_subsuming_call(subgoal_trie, subgoal_call) {
  match_mode = MATCH_EXACTLY
  path_type = VARIANT_PATH
  parent = trie_root(subgoal_trie)
  node = child(parent)
  var_chain = NULL
  push_arguments(term_stack, subgoal_call)

  while (!empty(term_stack))
    term = deref(pop(term_stack))
    push(term_log_stack, term)

    if (is_atom(term) or is_integer(term))
      match_node = match_constant_term(term, parent, node, var_chain, match_mode, path_type)
    else if (is_functor(term) or is_list(term))
      match_node = match_structured_term(term, parent, node, var_chain, match_mode, path_type)
    else if (is_variable(term))
      match_node = match_variable_term(term, parent, node, var_chain, match_mode, path_type)

    if (match_node == NULL)
      if (empty(call_choice_point_stack))   // no more alternatives
        return NO_PATH
      else
        match_mode = MATCH_TRIE_VARS   // backtrack mode
        (node, var_chain) = pop_call_choice_point_frame(call_choice_point_stack)
        parent = parent(node)
    else   // valid match, descend into node
      match_mode = MATCH_EXACTLY
      parent = match_node
      node = child(parent)

    return (path_type, parent)
}
```

Figure 4.3: Pseudo-code for procedure `lookup_subsuming_call`.

node by restoring all the auxiliary stacks state at the time of the call frame creation.

Each frame contains the next trie node to explore (`alt_node`), the current variable chain (`var_chain`), and the following stack indexes during frame creation: the top of the term stack (`term_stack_top`), the top of the term log stack (`term_log_stack_top`), and the top of the trail stack (`trail_stack_top`).

When popping a frame from the call choice point stack, the state of the auxiliary stacks and the other data structures is restored. Consider a computational state S_1 pushed onto the stack as frame $F1$. Given that we are at state S_2 and we need to backtrack to the previous state, first we need to remove $F1$ from the stack and then we use the frame's information to restore S_1.

Any terms that were popped from the term stack from S_1 to S_2, which were stored

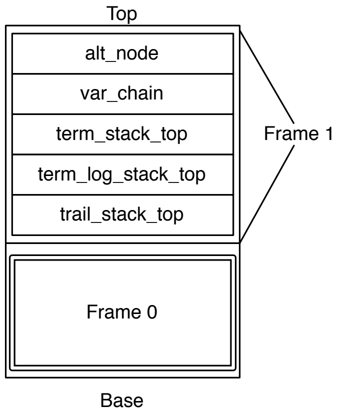


Figure 4.4: Call choice point stack organization.

in the term log stack must be restored back into the term stack. Then, any bindings made from S_1 to S_2 must be untrailed, which is accomplished by *unwinding* the trail stack. The unwind process untrails the trie variables bound to the variable bindings vector and the term variables which may have been made to point to the variable enumerator vector. The node associated with state S_2 and its successors are never visited again and the process continues until a leaf node is reached or the call choice point stack is exhausted.

4.1.2 Matching Constant Terms

The `match_constant_term` procedure (Figure 4.5) is called whenever the next term from the term stack is an integer or an atom term. First, the procedure checks if the match method is to exactly match the sub-term constant against a trie symbol, which means that this is the first time this trie level is explored (phase 1 in Figure 4.5). If the current trie level is represented by a simple linked list, both `node` and `var_chain` point to the start of the chain, but if the trie level is an hash table, `node` will be made to point to the corresponding indexed bucket and `var_chain` to the variable bucket, which is always the first bucket. This is accomplished by the `set_node_and_var_chain` procedure.

The procedure `find_matching_node` iterates over a linked list and locates a trie node that matches the **constant** symbol. When successful, we conditionally push a new call choice frame on the stack with the first node from the variable chain that contains a trie variable (`conditionally_push_call_choice_point_frame`). Note that only trie variables can be explored next as there is at most one node with the previously matched

```
match_constant_term(constant, parent, node, var_chain, match_mode, path_type) {
  // phase 1: try exact match
  if (match_mode == MATCH_EXACTLY)
    (node, var_chain) = set_node_and_var_chain(constant, node)
    match_node = find_matching_node(constant, node)
    if (match_node != NULL)
      conditionally_push_call_choice_point_frame(call_choice_point_stack, var_chain)
      return match_node
    else // no match found
      no_variant_found(parent, path_type)
      node = var_chain
  // phase 2: no exact match, try bound trie variables
  match_node = find_bound_trie_var(constant, node)
  if (match_node != NULL)
    push_call_choice_frame(call_choice_point_stack, sibling(match_node), var_chain)
    return match_mode
  // phase 3: no bound trie variable, try unbound trie variables
  match_node = find_unbound_trie_var(var_chain)
  if (match_node != NULL)
    bind_trie_var(match_node, constant)
    return match_node
  return NULL
}
```

Figure 4.5: Pseudo-code for procedure `match_constant_term`.

symbol.

If an exact match fails, we know that no variant path exists for the called subgoal, and we use the `no_variant_found` procedure to record the node from where the variant path can be constructed and to mark the `path_type` as `SUBSUMPTIVE_PATH`.

Phase 2 of `match_constant_term` can be reached by a failed exact match or by backtracking (remember that the match mode changes to `MATCH_TRIE_VARS` when backtracking). In this step we call `find_bound_trie_var` that will iterate over the `node` chain to look for *bound trie variables*. When traversing the subgoal trie, each new trie variable along a path is marked, so it is easy to check for old variables, which have already been bound to some term before arriving at the current node. The variable binding can be retrieved by checking the variable bindings vector for the position corresponding to this variable number, which points to an heap term. Given that we are trying to match a constant symbol, we verify if the bound term matches our symbol. In this case, we create a new choice point for the sibling node of the matched trie variable, and use the currently set variable chain.

Finally, if phases 1 and 2 fail, we try to match our constant against an unbound trie variable (procedure `find_unbound_trie_var`). Phase 3 can be also reached by a failed exact and bound trie variable match or by subsequent backtrack attempts. If a node is found, we trail the variable and its position on the variable bindings vector is made to point to the constant term (procedure `bind_trie_var`).

4.1.3 Matching Structured Terms

When a structured term appears on the term stack, either a functor or a list, the matching process works just like as for constants, except that the functor or list arguments are pushed to the term stack after an exact match is found (Figure 4.6).

```
match_structured_term(term, parent, node, var_chain, match_mode, path_type) {
  // phase 1: try exact match
  if (match_mode == MATCH_EXACTLY)
    (node, var_chain) = set_node_and_var_chain(term, node)
    match_node = find_matching_node(term, node)
    if (match_node != NULL)
      push_arguments(term_stack, term)
      conditionally_push_call_choice_point_frame(call_choice_point_stack, var_chain)
      return match_node
    else // no match found
      no_variant_found(parent, path_type)
      node = var_chain
  // phase 2: no exact match, try bound trie variables
  match_node = find_bound_trie_var(term, node)
  if (match_node != NULL)
    push_call_choice_frame(call_choice_point_stack, sibling(match_node), var_chain)
    return match_node
  // phase 3: no bound trie variable, try unbound trie variables
  match_node = find_unbound_trie_var(var_chain)
  if (match_node != NULL)
    bind_trie_var(match_node, term)
    return match_node
  return NULL
}
```

Figure 4.6: Pseudo-code for procedure `match_structured_term`.

Figure 4.7 shows the evolution of the term stack for finding a subsuming goal for subgoal `p(a,f(b))`. Step 1 shows the initial term stack, followed by the match of the term on top of the stack, `a`, against the trie node $n1$, `VAR0`. Being an unbound trie variable, the first position of the variable bindings vector is thus bound to `a`. Next, we try to match the functor `f/1` against node $n2$, but we fail (step 2). Then, node $n3$ succeeds, as it is an exact match, and the functor argument `b` is pushed on the term stack (step 3). Finally, `b` matches against node $n4$, and we find a subsuming call: `p(VAR0,f(b))`.

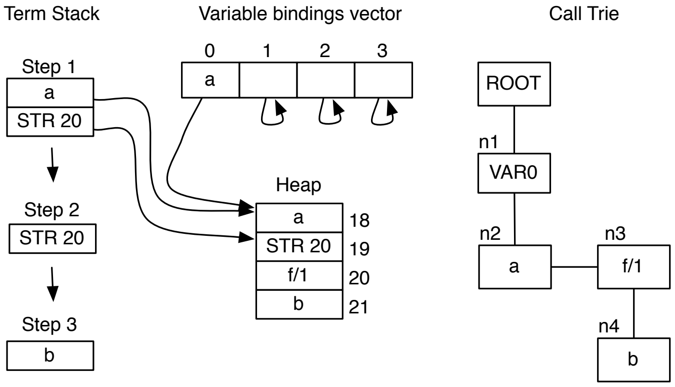


Figure 4.7: Finding a subsuming goal for subgoal `p(a,f(b))`.

4.1.4 Matching Variable Terms

The last case for the matching algorithm are variable terms. Because we are trying to find a more general goal, a term variable must only be matched or bound against a trie variable and a never seen variable must match only with an unbound trie variable.

Consider the case of trying to match `p(X,Y)` against `p(VAR0,VAR0)`. The first call variable `X` could match the first trie variable `VAR0`, but then the call variable `Y` must be matched against an unbound variable and thus the trie variable `VAR0` cannot be used, because it was already bound to variable `X`.

To recognize already seen call variables, we bind them to the variable enumerator vector, indexed by the corresponding trie variable number. The trail stack is used to trail those variables. When a bound variable must be matched again, first we try to pair it with the same trie variable, and if such trie variable can not be found, we try an unbound trie variable, thus, avoiding two different call variables to be bound to the same trie variable.

The pseudo-code for this procedure is shown in Figure 4.8. From it, we can conclude that a variant path cannot be found when: (1) we cannot match an already seen call variable against the same trie variable already bound to it; or (2) a new trie variable cannot be found for a new call variable.

```
match_variable_term(variable, parent, node, var_chain, match_mode, path_type) {
  if (match_mode == MATCH_EXACTLY)
    (node, var_chain) = set_node_and_var_chain(variable, node)
    if (!is_in_variable_enumerator_vector(variable))
      // variable not seen before
      foreach (match_node in node)
        if (is_trie_var(match_node) and is_new_variable(match_node))
          // only one new trie variable per level, no choice point needed
          bind_trie_var(match_node, variable)
          mark_variable_enumerator_vector(variable, var_index(match_node))
          return match_node
      no_variant_found(parent, path_type)
      return NULL
  // variable has been seen before
  foreach (match_node in node)
    if (is_trie_var(match_node) and !is_new_variable(match_node))
      if (identical_terms(trie_var_bindings[match_node], variable))
        push_call_choice_frame(call_choice_point_stack, sibling(match_node), var_chain)
        return match_node
  // variant path is not possible here
  no_variant_found(parent, path_type)
  // match against unbound trie variable
  foreach (match_node in var_chain)
    if (is_trie_var(match_node) and is_new_variable(match_node))
      // only one new trie variable per level, no choice point needed
      bind_trie_var(match_node, trie_var_bindings[prolog_var_index(variable)])
      return match_node
  return NULL
}
```

Figure 4.8: Pseudo-code for procedure `match_variable_term`.

4.1.5 Variant Continuations

A *variant continuation* is built when the algorithm detects that a variant path of the called subgoal cannot be found on the subgoal trie during the search process. A variant continuation stores all the needed information to later resume the algorithm that creates the variant path. This includes the node from where the rest of the variant path is created (i.e., the last node from where the search for a variant path was still valid), the term stack and all the bindings made to the call variables that were trailed on the trail stack. In previous pseudo-code listings we used the procedure `no_variant_found` to do that. This procedure creates a variant continuation the first time it is called.

Figure 4.9 shows a variant continuation that is built after the match process failed at node `c`. If a variant path for subgoal `p(X,f(Y),b)` needs to be created, the following must be done: (1) the term stack is restored with the terms saved on the continuation;

(2) the trail stack is initialized with the two variable heap addresses and each variable is bound to the saved enumerator addresses. The variable enumerator vector is used during the insertion of variant paths to detect if a variable was already seen and easily compute its number by looking at the enumerator position.

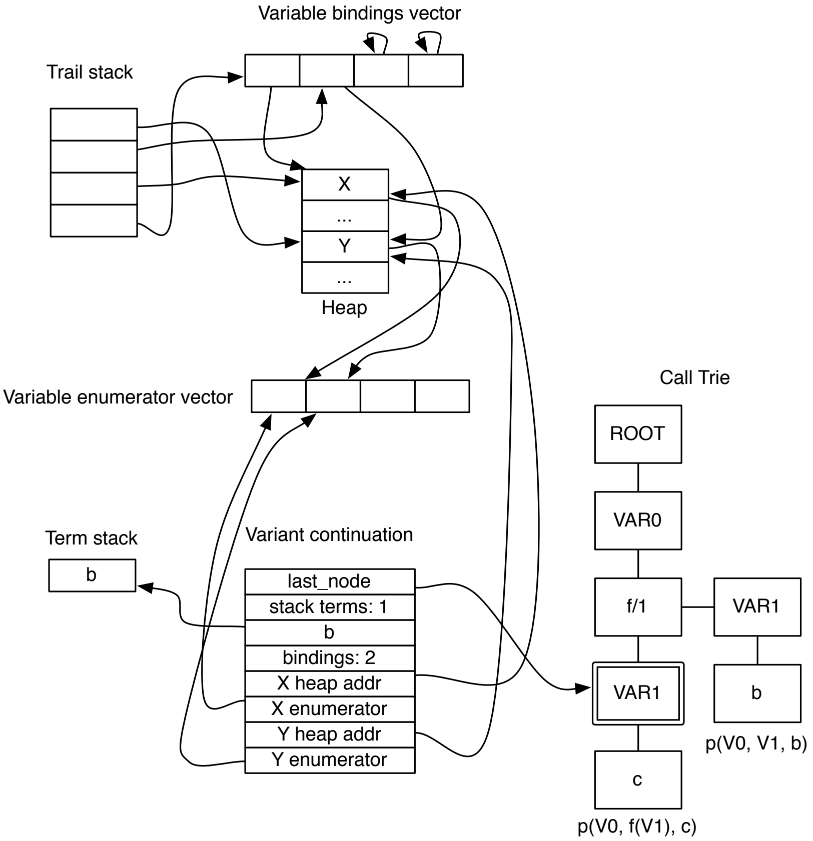


Figure 4.9: Variant continuation for subgoal `p(X,f(Y),b)`.

4.2 Answer Templates

In a variant engine, the substitution factor [RRS$^+$95] represents the variables which exist in the terms of the argument registers. These variables are dereferenced when inserting answers in an answer trie (for general calls) or bound to terms when consuming answers from an answer trie (for consumer calls). Figure 4.10 shows how a substitution factor is constructed on the local stack associated with a generator choice point. Note that the substitution factor size (2) in the example is also stored.

When using call by subsumption, the same factor is used for subgoals which do not

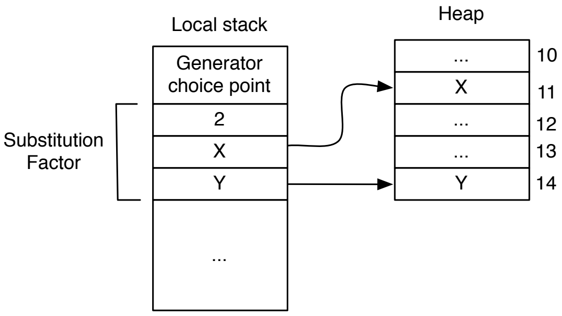


Figure 4.10: Substitution factor for `p(X,f(Y))`.

consume from more general subgoals, i.e., generator subgoals or variants of generator subgoals. We call this type of substitution factor a *generator answer template.*

For subgoals with more general subgoals, subsumed subgoals or variants of subsumed subgoals, the answer template must specialize the generator answer template from the most general subgoal, and is called a *consumer answer template.* Consumer answer templates have the exact same size of their corresponding generator answer templates and instead of being composed only with variables, they can also include other types of subterms.

Consider a generator subgoal `p(X,f(Y))` and that the subgoal `p(a,f(g(2,X)))` is then called. The corresponding answer template `[a, g(2,X)]` is derived by instantiating the generator variables, `X` and `Y`, with the terms in the consumer subgoal (see Figure 4.11).

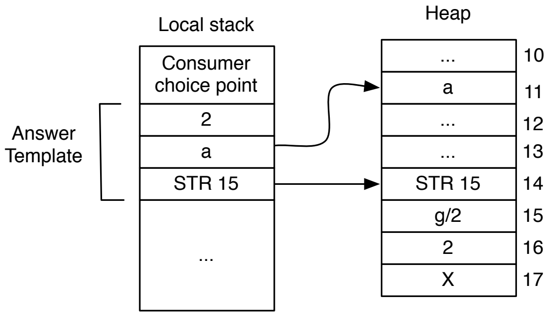


Figure 4.11: Consumer answer template for `p(a,f(g(a,X)))`.

The construction of answer templates is done when searching the subgoal trie for a more general goal. If no subsuming subgoal is found, then a generator answer template is built. If a subsuming subgoal exists, then the found subgoal frame S can be either:

1. a generator subgoal frame: the answer template is built using the variable bindings vector as shown in procedure **construct_answer_template_from_lookup** (Figure 4.12);

2. a consumer subgoal frame: the answer template is reconstructed by using the generator subgoal frame of S as we consume only from proper generators. The procedure `construct_answer_template_from_generator` (Figure 4.13) builds this answer template by matching new trie variables against terms from the specific subgoal. It uses another stack, the *symbol stack*, to push the trie symbols from the general subgoal path. The term stack is used to push the called subgoal arguments that will be matched against the trie symbols. Note that when a functor or list symbol appears on the symbol stack the current term is also certainly a functor or list, because the called subgoal specializes the more general subgoal.

```
construct_answer_template_from_lookup() {
  total = 0
  foreach (binding in variable_bindings_vector)
    total++
    answer_template[total] = binding
  answer_template[0] = total
  return answer_template
}
```

Figure 4.12: Pseudo-code for procedure `construct_answer_template_from_lookup`.

```
construct_answer_template_from_generator(subgoal_call, generator_sf) {
  push_trie_path(symbol_stack, subgoal_trie_path(generator_sf))
  push_arguments(term_stack, subgoal_call)

  total = 0
  while (!empty(term_stack))
    term = deref(pop(term_stack))
    symbol = pop(symbol_stack)
    if (is_trie_var(symbol) and is_new_variable(symbol))
      total++
      answer_template[total] = term
    else if (is_functor(symbol) or is_list(symbol))
      push_arguments(term_stack, term)

  answer_template[0] = total
  return answer_template
}
```

Figure 4.13: Pseudo-code for procedure `construct_answer_template_from_generator`.

4.3 Time Stamped Answer Trie

A time stamped node extends an answer trie node with time stamp information. Each node contains the following fields: **symbol**, **child**, **parent**, **sibling** and **timestamp**. For implementation and algorithmic purposes, each node also needs a bit field, **status**, that defines some node properties that will be described shortly.

Insertion of an answer S into a TST can be divided into two steps:

1. Finding a more general answer S' on the trie.

2. Inserting S if S' could not be found.

To implement step 1, we use the same algorithm described in Section 4.1 but now to search for a *more general answer* or a *repeated answer* (i.e., a variant of S). If step 1 fails, step 2 then uses the variant continuation to resume the insertion of the answer on the last node where the variant answer path was still expected to be found during step 1.

```
subsumptive_answer_search(trie_root, ans_vector)
  (path, leaf) = lookup_subsuming_call(trie_root, ans_vector) // step 1
  if (path == NO_PATH)
    // step 2
    node = restore_variant_continuation()
    leaf = tst_insert(trie_root, node)
  return leaf
}
```

Figure 4.14: Pseudo-code for procedure subsumptive_answer_search.

The procedure subsumptive_answer_search (Figure 4.14) implements this process and needs two arguments: the root of the answer trie trie_root, and the answer as a vector of terms, ans_vector.

When a chain of sibling nodes becomes larger than a threshold value, we dynamically index the nodes through an hash table to provide direct node access and therefore optimize the search. Given that an hash table can have a large number of answer nodes with different time stamp values, we maintain a reference to these nodes, in decreasing order of their time stamp values in a double linked list called the *time stamped index*. Besides the double linked list pointers, each index node contains a pointer to the corresponding answer node and the value of its time stamp. Each answer node indexed in the hash table has a different use for the **timestamp** field: instead of containing a positive integer, contains a pointer to the respective index node. The trie node field **status** distinguishes both cases. Figure 4.15 illustrates an hash table with a time stamp index.

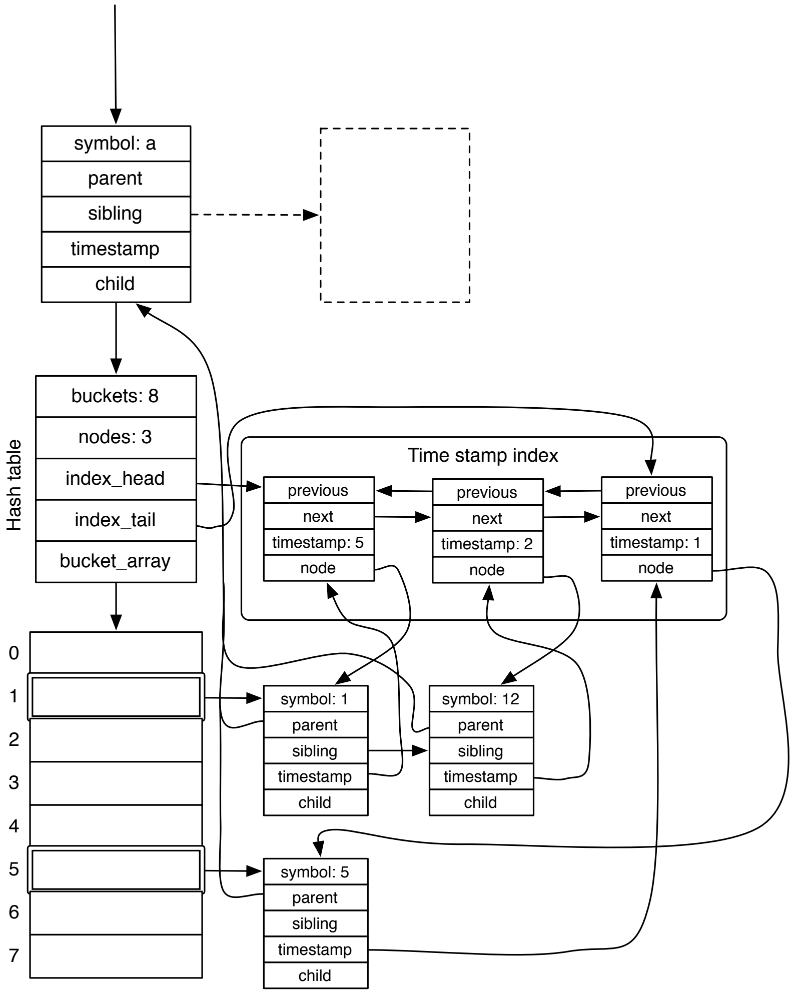


Figure 4.15: Indexing nodes through an hash table with time stamp indexes.

4.3.1 Inserting New Answers

Once the variant continuation is restored, the rest of the answer path can be inserted on the trie starting from the restored node. Inserting is then a simple operation because no checking for equal trie symbols is required. The first symbol will be inserted either into: (1) a childless parent node (i.e., the trie root); (2) a parent node pointing to a sibling chain list; (3) a parent node pointing to an hashed child node. Every other symbols will be inserted on childless parents.

Procedure `tst_insert` (Figure 4.16) does the job of inserting all the terms contained in a stack of terms into the trie, starting from `node`. The difference between procedures `tst_add_symbol` and `tst_insert_symbol` is that the former inserts symbols on childless nodes and the later on nodes with children.

```
tst_insert(trie_root, node) {
  symbol = process_term_stack(term_stack)
  if (child(node) == NULL)
    // inserting on the root
    node = tst_add_symbol(node, symbol)
  else if (is_hash_table(node))
    node = tst_hash_table_add_symbol(node, symbol)
  else
    node = tst_insert_symbol(node, symbol)

  // at this point, just add nodes on childless parents
  while (!empty(term_stack))
    symbol = process_term_stack(term_stack)
    node = tst_add_symbol(node, symbol)

  update_timestamps(trie_root, node)
  return node
}
```

Figure 4.16: Pseudo-code for procedure `tst_insert`.

The procedure `process_term_stack` pops a term from the term stack and converts the term to a trie representation, which is usually called a *symbol*. If the term is a functor or a list, the arguments are pushed into the stack of terms to be processed next.

While appending a new node in a sibling list does not involve any time stamp indexing (see Figure 4.17), inserting a new node into an hash table does, because hash tables index nodes by time stamp. Figure 4.18 illustrates the insertion of a symbol (`25`) and the creation of the respective index node. Note that the **index_head** field was changed to point to the new index node, which is always the node with the greatest time stamp.

4.3.2 Updating Time Stamps

Once an answer is inserted, the time stamps must be updated. The time stamp for the new answer is calculated by inspecting the time stamp of the trie root node. Next,

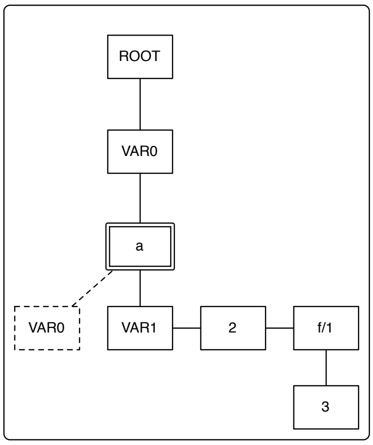


Figure 4.17: Inserting answer {VAR0,a,VAR0}.

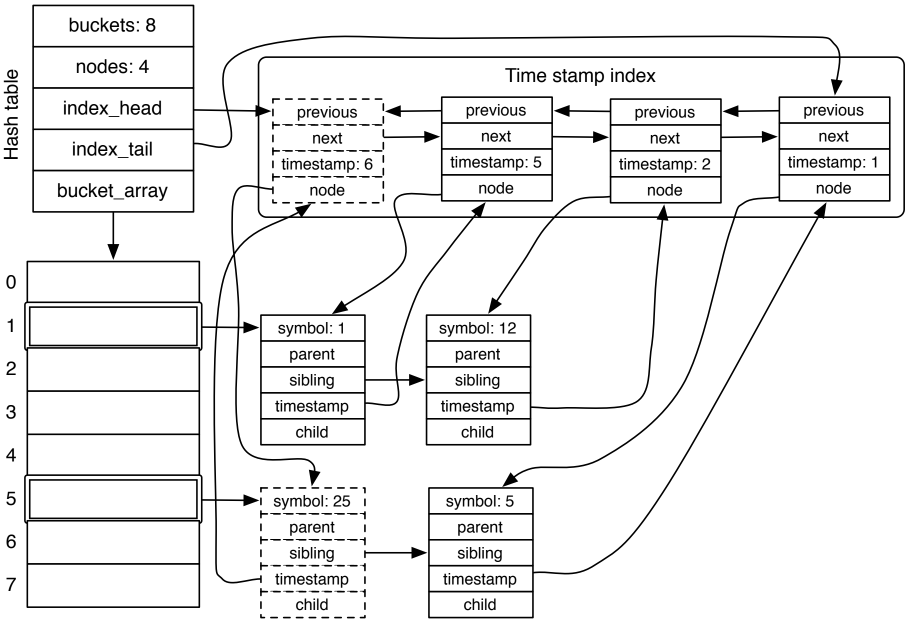


Figure 4.18: Inserting a node into the hash table and updating the index.

we update the time stamps in the answer trie branch starting from the answer leaf node to the root node.

If the current node is an hashed node with time stamp indexes, its index node is moved to the head of the index's double linked list. We can test whether an answer node is indexed by an hash table by inspecting its **status** field. Figure 4.19 contains the pseudo-code for procedure `update_timestamps`.

When moving an index node N, the **index_head** field of the hash table must be

```
update_timestamps(trie_root, leaf) {
  new_timestamp = timestamp(trie_root) + 1

  while (leaf != trie_root)
    if (is_hashed_node_with_time_stamp_indexes(leaf))
      // relocate index node
      promote_entry(leaf, new_timestamp)
    else
      timestamp(leaf) = new_timestamp
    leaf = parent(leaf)

  timestamp(trie_root) = new_timestamp
}
```

Figure 4.19: Pseudo-code for procedure `update_timestamps`.

updated to point to N. Moreover, if N was at the end of the chain, the **index_tail** field must be also updated to the **previous** field of N. Previous pointers of the **next** node of N and the next pointer of the **previous** node of N must also be updated to keep the chain consistent. Figure 4.20 illustrates the relocation of an index node, after its time stamp be updated from 5 to 7.

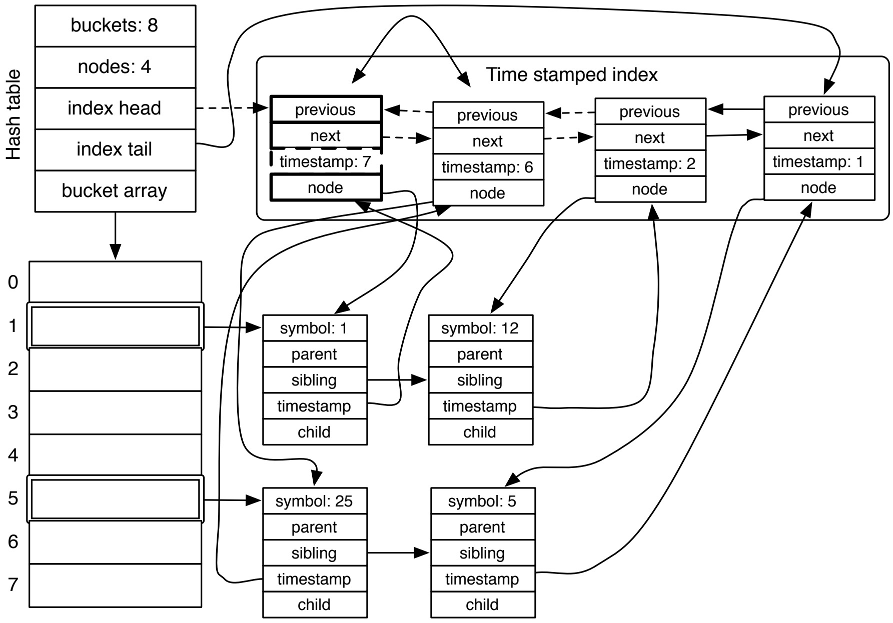


Figure 4.20: Promoting an index node.

4.3.3 Lazy Creation of Time Stamp Indexes

Time stamped indexes are only created when a consumer subgoal is first called. We must iterate over all the hash tables present on the trie to create the time stamp index. To efficiently locate all hash tables in an answer trie, we chain these hash tables using the **next** field and the start of this chain is stored in the **sibling** field of the root node.

Creating the index for an hash table amounts to iterating over the hashed nodes and orderly inserting new index nodes on the index chain. Later, when the subgoal completes, the time stamp indexes can be thrown away to save space, because they are only necessary during collection of relevant answers for consumer subgoals.

4.4 Collecting Relevant Answers

The process of collecting relevant answers for a consumer subgoal G from an answer trie T of the generator subgoal G', involves searching T for a set S of answers that unify with the consumer answer template AT and are newer than the time stamp TS stored in the consumer subgoal frame. After collection, TS is updated to the time stamp of the root node of T, thus avoiding repeated answers in future iterations of the algorithm. Collected answers are then appended to the list of answers of the consumer subgoal frame, so that they can be reused in future calls of G.

Various data structures are used for this algorithm, namely:

- *WAM data structures*: the push down list (PDL), heap, trail, and associated registers. The heap is used to build structured terms, in which the answer template or trie variables are bound. Whenever a variable is bound, we trail it using the WAM trail. The *unify* operation provided by the WAM is used to check for term equality in structured terms;

- *term stack*: used to store the next terms to be processed as we navigate through the time stamped trie T;

- *term log stack*: when an unification fails, there is a need to backtrack to inspect other alternative branches. This stack is used to store already processed terms of the term stack, so they can be restored back during backtracking;

- *variable bindings vector*: stores the bindings for the trie variables;

- *choice point stack*: stores choice point frames, where each frame contains information needed to restore the computation in order to search for alternative branches.

The pseudo-code for the algorithm is presented in Figure 4.21. The whole algorithm can be summarized into eight steps:

1. Setup phase: setup term stack and WAM machinery.
2. Fetch a term T from the term stack;
3. Search for a node N at the current trie level that has a valid time stamp;
4. Search for the next valid node to be pushed on the choice point stack;
5. Unify T with the trie symbol of N;
6. Proceed into the child of N or, if steps 3 or 5 fail, backtrack by popping a frame from the choice point stack and use the alternative node to unify;
7. Once a leaf is reached, mark it as a new answer and possibly backtrack to retrieve more answers.
8. If no more choice point frames exist, return the marked answers.

The setup phase pushes the answer template into the term stack and backups the WAM registers. The trail register is set to the next free position of the WAM trail, thus avoiding writing on frozen segments. Registers HB, H and TR are saved as they will be manipulated and need to be restored to avoid any interference with the normal WAM execution.

4.4.1 Choice Point Stack

The choice point stack stores alternative search branches to use upon backtracking. Each stack frame (see Figure 4.22) stores the following fields: `alt_node`, the alternative node to explore; `term_stack_top`, the top of the term stack; `term_log_stack_top`, the top of the term log stack; `trail_top`, the current trail position; and `saved_HB`, the register HB.

```
tst_collect_relevant_answers(trie_root, ts, answer_template) {
  answers = NULL
  push_terms(term_stack, answer_template)
  save_wam_registers()
  parent = trie_root
  node = child(parent)

while_loop:
  while (!empty(term_stack))
    term = deref(pop(term_stack))

    if (is_atom(term) or is_integer(term))
      unify_node = unify_constant_term(term, node, ts)
    else if (is_functor(term) or is_list(term))
      unify_node = unify_structured_term(term, node, ts)
    else if (is_variable(term))
      unify_node = unify_variable_term(term, node, ts)

    if (unify_node != NULL)
      parent = unify_node
      node = child(parent)
      continue
    else if (empty(choice_point_stack))
      unwind_wam_trail()
      restore_wam_registers()
      return answers
    else
      node = pop_choice_point_frame(choice_point_stack)
      parent = parent(node)

  new_answer_found(parent, answers)
  if (empty(choice_point_stack)
    unwind_wam_trail()
    restore_wam_registers()
    return answers
  node = pop_choice_point_frame(choice_point_stack)
  parent = parent(node)
  goto while_loop
}
```

Figure 4.21: Pseudo-code for procedure `tst_collect_relevant_answers`.

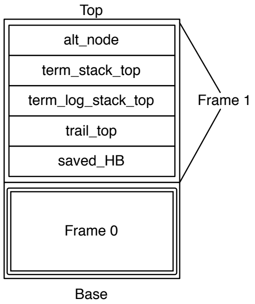


Figure 4.22: Choice point stack organization.

The HB register serves the same purpose as the standard HB register in a WAM choice point. During execution, the WAM's HB register is compared against the value of H to determine if a variable is *conditional*, that is, if a variable needs to be trailed, so that when execution backtracks to a previous choice point we can reset variable bindings. In our case, instead of executing WAM code, we unify a time stamped node, thus the meaning of a conditional variable is extended to include the trie variables in the variable bindings vector.

When a choice point frame is popped from the stack, the state of the computation is resumed by executing the following actions:

- the current node and parent node are reset;
- all terms stored in the term log stack are pushed back to the term stack;
- the trail is unwinded to reset the variables that were bound after choice point creation;
- registers H and HB are also reseted to previous values.

4.4.2 Unification of Constant Terms

Once a trie node N is reached we must select the next trie node N' that unifies with our term and has a valid time stamp. Node N can lead either to a simple node chain or an hash table. With constant terms we can index the hash table to prune the search space (procedure `set_match_and_unify_chains` in Figure 4.23) by using the *match bucket*.

Because variables can unify with the constant term, there is a need to retrieve the variable chain (from the variable bucket), which will be used as the alternative chain to push on the choice point stack. We call this chain the *unify chain*. If the constant is found on the match chain, the unify chain is used as alternative, but if no match was found, the variable chain will be attempted next and, depending on the remaining nodes, also be used as the backtracking alternative.

In Figure 4.23 we present the pseudo-code for procedure `unify_constant_term`. First, we check for an hash table and inspect the match bucket using `search_chain_exact_match` (Figure 4.24). Next, if no match is found we execute `search_chain_unify_with_constant` (Figure 4.25) on the unify chain. Otherwise, if no hash table is found, we would

```
unify_constant_term(constant, node, ts) {
  if (is_hash_table(node))
    // retrieve the indexed and variable buckets
    (match_chain, unify_chain) = set_match_and_unify_chains(constant, node)
    if (match_chain != unify_chain)
      node = search_chain_exact_match(constant, match_chain, unify_chain, ts)
      if (node != NULL)
         return node
      // exact match failed
      node = unify_chain
    if (node == NULL)
      return NULL
  return search_chain_unify_with_constant(constant, node, ts)
}
```

Figure 4.23: Pseudo-code for procedure `unify_constant_term`.

```
search_chain_exact_match(term, match_chain, unify_chain, ts) {
  foreach (node in match_chain)
    if (term == symbol(node))
      if (valid_timestamp(timestamp(node), ts))
        push_choice_point_frame(choice_point_stack, next_valid_node(unify_chain, ts))
        push(term_log_stack, term)
        return node
      else
        return NULL
  return NULL
}
```

Figure 4.24: Pseudo-code for procedure `search_chain_exact_match`.

consider the simple chain of sibling nodes as the unify chain and simply execute `search_chain_unify_with_constant` on it.

When searching the unify chain, first we locate the next node with a valid time stamp on the chain, that is, with the time stamp greater than our target time stamp. Next, we *dereference* the node symbol by using `trie_deref`, which returns a position on the variable bindings vector if the node symbol is a trie variable, hence allowing trie variables to be used as *normal* variables.

In case (1), a variable was found, which can be a position on the variable bindings vector or a Prolog variable that was bound to a trie variable. Either way, we bind the variable to the constant term, push a new choice point frame with the next valid node, and return `chain` to be explored next. Please note that the procedure `bind_and_conditionally_trail` tests if the variable (first argument) is a conditional variable and then trails it using the WAM trail.

```
search_chain_unify_with_constant(constant, chain, ts) {
  chain = next_valid_node(chain, ts)
  while (chain != NULL)
    alt_chain = chain_next_valid_node(sibling(chain), ts)
    symbol = trie_deref(symbol(chain))
    if (is_variable(symbol)) // case (1)
      push_choice_point_frame(choice_point_stack, alt_chain)
      bind_and_conditionally_trail(symbol, constant)
      push(term_log_stack, constant)
      return chain
    else if (symbol == constant) // case (2)
      // exact match
      push_choice_point_frame(choice_point_stack, alt_chain)
      push(term_log_stack, constant)
      return chain
    else
      chain = alt_chain
  }
  // case (3)
  return NULL
}
```

Figure 4.25: Pseudo-code for procedure `search_chain_unify_with_constant`.

In case (2) the node symbol matches our constant and we simply push a new choice point frame and advance into the next node.

Otherwise, if we could not found a valid trie node in the unify chain, we get into case (3), and the choice point stack must be used to try alternatives.

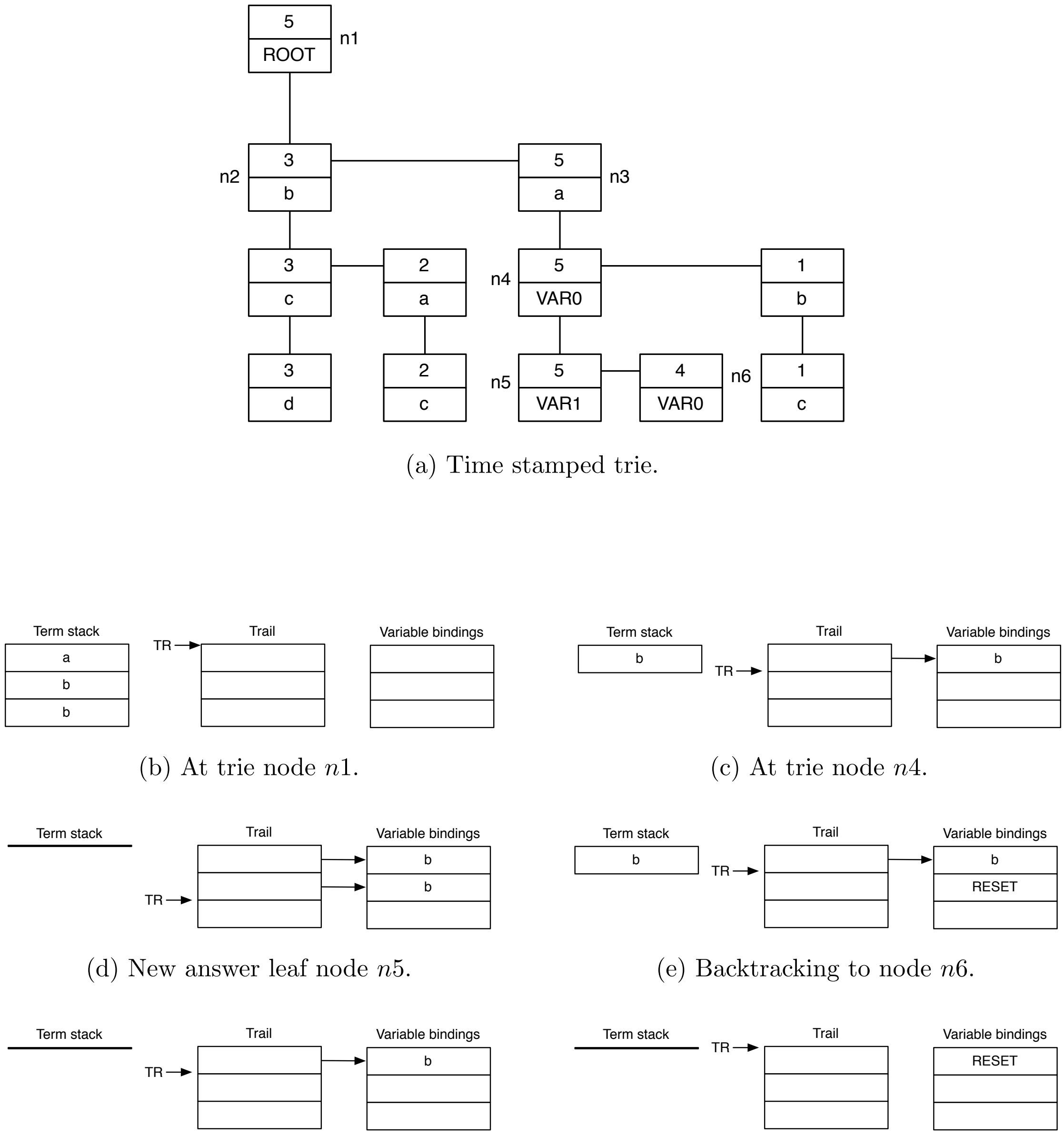
As an example, let's consider the time stamped trie in Figure 4.26a. The input answer template is {`a,b,b`} and the target time stamp is 3.

We start on node $n1$, the root of the trie (Figure 4.26). The unify chain is composed by nodes $n2$ and $n3$. Node $n2$ is discarded because its time stamp is invalid, but node $n3$ has a valid time stamp and its symbol matches `a` (Figure 4.26b).

On node $n3$, only our first alternative, node $n4$, has a valid time stamp and, after doing `trie_deref` we find an unbound variable, `VAR0`, which is represented by the first position of the variable bindings vector. This variable is trailed and bound to `b`, resulting in what is presented in Figure 4.26c.

On node $n4$, the unify chain is composed by nodes $n5$ and $n6$. Both have valid time stamps (> 3). Node $n5$ is attempted first and easily unifies, because it is an unbound trie variable. Leaf node $n5$ is our first answer (Figure 4.26d).

Now, we need to backtrack to collect more answers. The top choice point frame is



(a) Time stamped trie.

(b) At trie node $n1$.

(c) At trie node $n4$.

(d) New answer leaf node $n5$.

(e) Backtracking to node $n6$.

(f) New answer as the leaf node (f).

(g) Untrailing variables and returning.

Figure 4.26: Unification of answer template {`a, b, b`} with time stamp 3.

retrieved from the stack resulting in a variable being untrailed and the term `b` being pushed into the term stack (Figure 4.26e).

In node $n6$ we dereference the trie variable `VAR0` and get the constant term `b`, which matches the target term. No binding or trailing is needed and we succeed in collecting another relevant answer (Figure 4.26f).

As there are no more available choice points we need to untrail any bindings made and return the answers found (Figure 4.26g), finishing the search.

4.4.3 Unification of Structured Terms

For structured terms, the unification process is similar to constant unification (Figure 4.27). First, we check if the current trie node is an hash table and then the match and unify chains are computed. If the match chain contains a valid trie node, before we descend into the child node we must push the functor or list arguments into the term stack, so they can be unified with the next trie nodes.

```
unify_structured_term(term, node, ts) {
  if (is_hash_table(node))
    // retrieve the indexed and variable buckets
    (match_chain, unify_chain) = set_match_and_unify_chains(term, node)
    if (match_chain != unify_chain)
      node = search_chain_exact_match(term, match_chain, unify_chain, ts)
      if (node != NULL)
         push_arguments(term_stack, term)
         return node
      // exact match failed
      node = unify_chain
    if (node is NULL)
      return NULL
  return search_chain_unify_with_structured_term(term, node, ts)
}
```

Figure 4.27: Pseudo-code for procedure `unify_structured_term`.

When using the unify chain (procedure `search_chain_unify_with_structured_term`), we also iterate the chain looking for valid time stamped nodes. For each valid node four situations may arise (Figure 4.28):

1. The trie symbol is a variable, which is trailed and bound to the structured term;
2. The trie symbol is a structured term and matches our functor or list. The term arguments are pushed into the term stack for unification;

3. We find a trie variable bound to a structured term. The WAM function `unify` is executed to check for a match and perform additional unifications;

4. No match was found, the next alternative node is inspected.

```
search_chain_unify_with_structured_term(term, chain, ts) {
  chain = next_valid_node(chain, ts)
  while (chain != NULL)
    alt_chain = next_valid_node(sibling(chain), ts)
    symbol = trie_deref(symbol(chain))
    if (is_variable(symbol)) // case (1)
      push_choice_point_frame(choice_point_stack, alt_chain)
      bind_and_conditionally_trail(symbol, term)
      push(term_log_stack, term)
      return chain
    else if (is_functor(symbol) or is_list(symbol))
      if ((is_functor(symbol(chain)) or is_list(symbol(chain))) and symbol == term)
        // case (2)
        push_choice_point_frame(choice_point_stack, alt_chain)
        push(term_log_stack, term)
        push_arguments(term_stack, term)
        return chain
      else if (unify(term, symbol)) // case (3)
        // trie variable bound to an heap structured term
        push_choice_point_frame(choice_point_stack, alt_chain)
        push(term_log_stack, term)
        return chain
    else
      chain = alt_chain
  }
  // case (4)
  return NULL
}
```

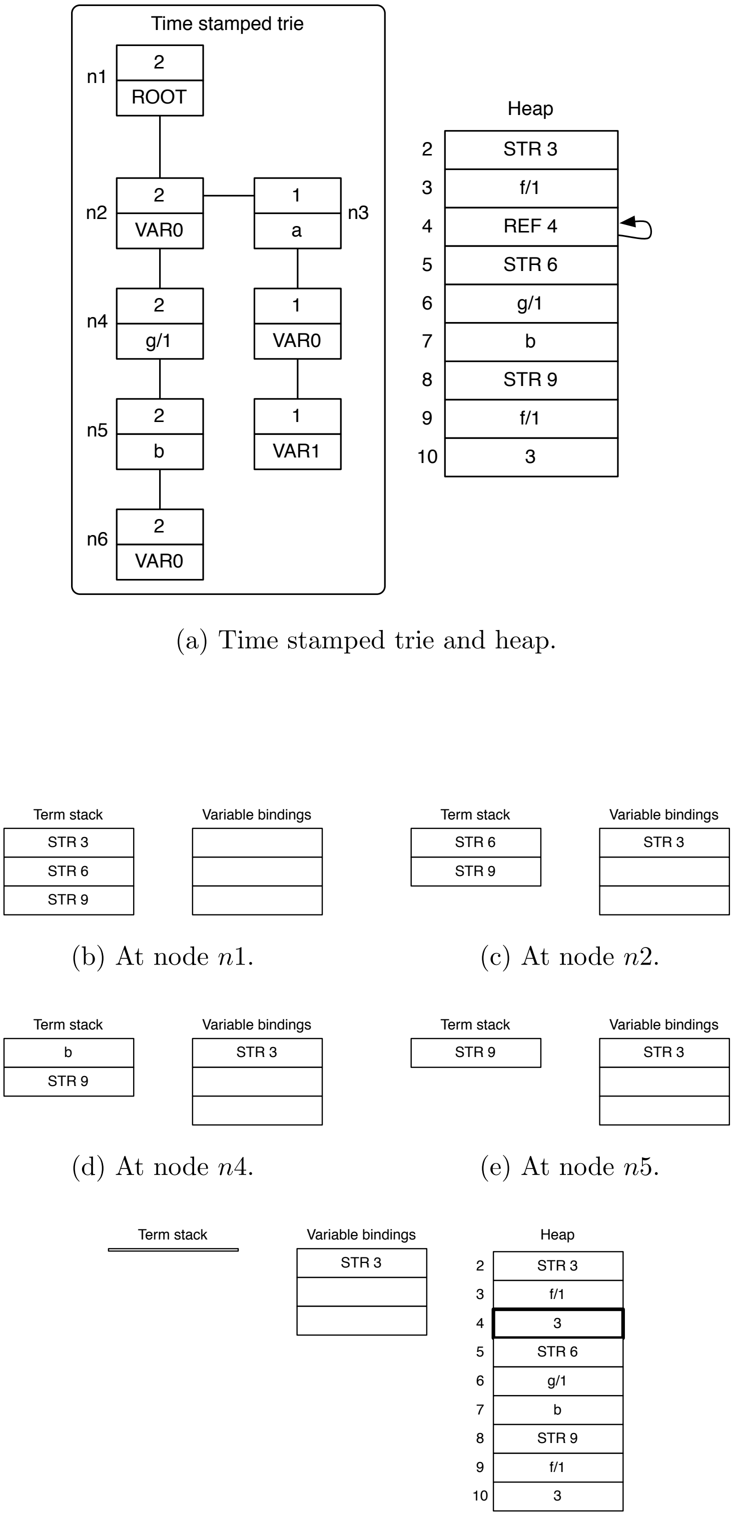
Figure 4.28: Pseudo-code for procedure `search_chain_unify_with_structured_term`.

Let's consider the time stamped trie in Figure 4.29a and the following answer template: {`STR 3, STR 6, STR 9`}. The target time stamp is 1.

Initially, at node $n1$, the term stack contains the full answer template and the variable bindings vector is empty (Figure 4.29b). Here, only node $n2$ satisfies the time stamp requirements ($2 > 1$).

Node (b) contains a trie variable and the current term is `STR 3` or `f(VAR)`. In this situation, the variable bindings position for **VAR0** is trailed and bound to `STR 3` (Figure 4.29c).

Next, node $n4$ contains the symbol `g/1` and the current term is `STR 6` or `g(b)`, which matches. The argument `b` of `g(b)` is thus pushed into the term stack to be processed



(a) Time stamped trie and heap.

(b) At node $n1$.

(c) At node $n2$.

(d) At node $n4$.

(e) At node $n5$.

(f) At node $n6$.

Figure 4.29: Unification of answer template {`f(X), g(b), f(3)`} with time stamp 1.

in the next node (Figure 4.29d).

Then, node $n5$ has the symbol `b`, which matches with `b` from the term stack, and execution proceeds to node $n6$ (Figure 4.29e).

At $n6$ we find a trie variable, which, after being dereferenced, contains the functor `f(VAR)`. The current term to be unified is `f(3)`. In this situation we call `unify`, which will try to unify both terms. The unification has the side effect of setting the heap variable cell 4 to `3` (Figure 4.29f). This variable is conditional because it is positioned before the register `HB`, which given the algorithm must be greater than 10.

4.4.4 Unification of Variable Terms

Variable unification is done when the next term to unify is a variable. In this case, trie branches are only pruned by using the time stamp as variables can unify with anything. Figure 4.30 presents the procedure `unify_variable_term`. In this function, three situations may arise:

1. The current node is an hash table. In this situation we can select the next transitions by using the time stamped index, which can efficiently prune based on the time stamp. Notice that we visit an hash table only once, subsequent backtracking uses the time stamp index nodes;

2. The current node is inside an hash table and thus indexed on the time stamp index. The node is matched against the variable and the alternative node is selected by following the index chain link;

3. Current node is a simple sibling chain. Both the chain and the alternative chain are set by iterating over the node chain, looking for valid time stamps.

Once the chains are set, we create a new choice point, run the variable unification algorithm and proceed into the next trie node. From the pseudo-code in Figure 4.31, unification with a term variable is dictated by the type of trie symbol. It follows the following rules:

- Constant: the term variable is bound to the symbol and conditionally trailed;
- Structured term: if the symbol was a trie variable bound to a term then we bind the variable to the heap location; else, we create a new structure (functor or list)

```
unify_variable_term(variable, node, ts) {
  if (is_hash_table(node))
     // case 1: current node is an hash table
    index = index_head(node)
    if (timestamp(index) > ts)
      node = node(index)
      alt_chain = node(next_valid_index_node(index, ts))
    else
      return NULL
  else if (is_hashed_node(node))
    // case 2: current node is an hashed node and has a corresponding index node
    // can only be here via backtracking
    alt_chain = node(next_valid_index_node(index_node(node), ts))
  else
    // case 3: simple chain of siblings
    node = chain_next_valid_node(node, ts)
    if (node == NULL)
      return NULL
    alt_chain = sibling(node)

  push_choice_point_frame(choice_point_stack, alt_chain)
  push(term_log_stack, variable)
  symbol = trie_deref(symbol(node))
  return unify_with_variable(variable, symbol, node)
}
```

Figure 4.30: Pseudo-code for procedure `unify_variable_term`.

on the heap and bind the variable to it, resulting in a term with various heap variables as arguments, which will be pushed into the term stack and will be used in the next iterations of the algorithm;

- Variable: if the variable is a trie variable, we bind and trail it; if it is an heap variable that was dereferenced from a trie variable using `trie_deref`, `unify` chooses the binding direction, resulting in one of the variables being trailed.

As an example, consider the trie and heap in Figure 4.32a. The input answer template is {`REF 2`,`REF 2, b`} and the target time stamp is 2.

On root node $n1$, we start with the configuration presented in Figure 4.32b.

Node $n2$ is then the only valid transition, with time stamp 3. The functor `f/1` is unified against the variable `REF 2`, which results in the functor `f/1` being created on the heap and its argument (`REF 5`) being pushed into the term stack (Figure 4.32c).

The yet unbound functor argument matches atom `a` in node $n4$, resulting in the update of the heap cell 5 (Figure 4.32d).

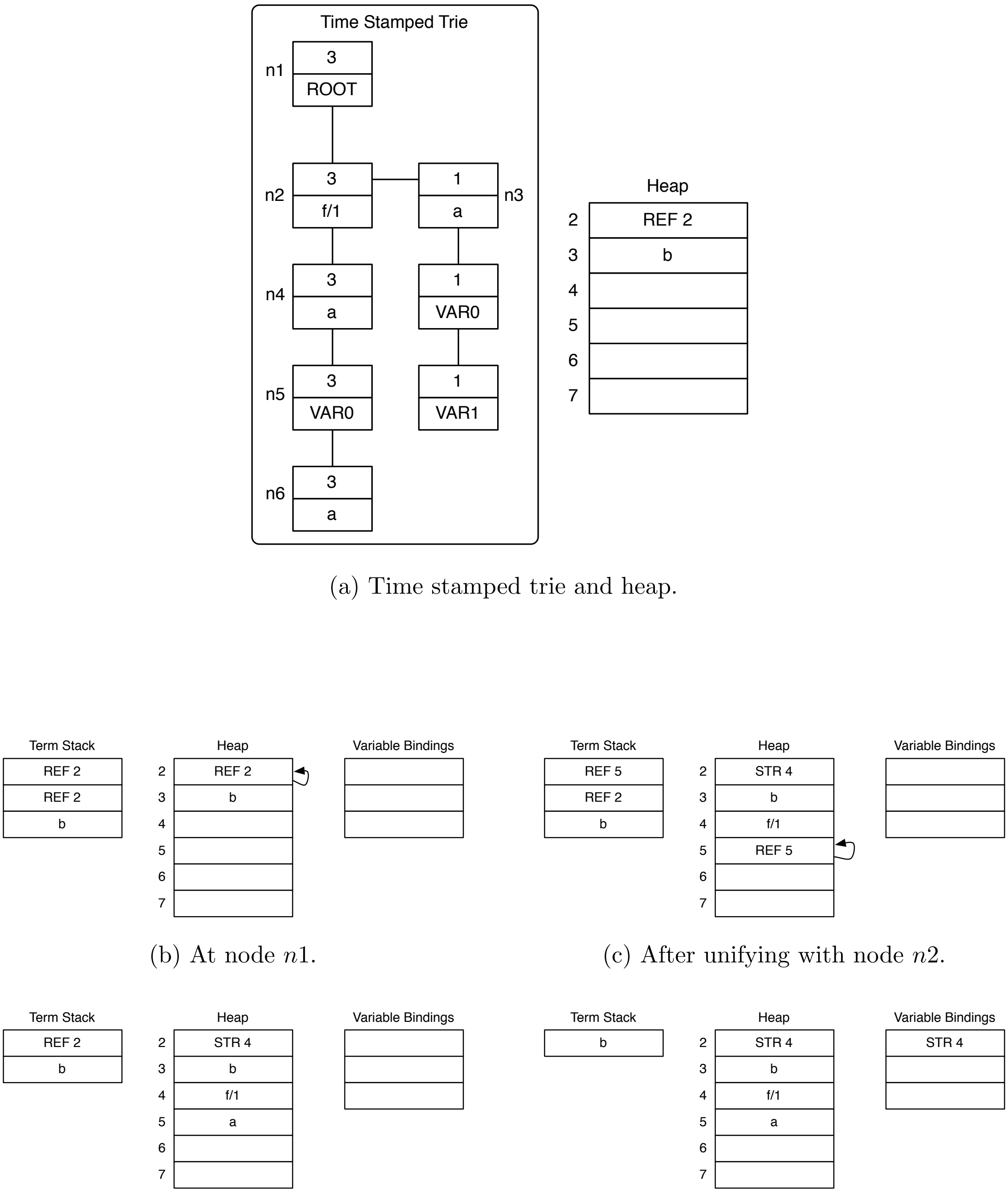
```
unify_with_variable(variable, symbol, node) {
  if (is_constant(symbol))
    bind_and_conditionally_trail(variable, symbol)
  else if (is_functor(symbol) or is_list(symbol))
    if (is_list(symbol(node)) or is_functor(symbol(node))
      term = create_heap_structure(symbol)
      bind_and_conditionally_trail(variable, term)
      // push new structure arguments
      push_arguments(term_stack, deref(variable))
    else
      // trie variable bound to an heap structure
      bind_and_conditionally_trail(variable, symbol)
  else if (is_variable(symbol))
    if (is_trie_variable(symbol))
      bind_and_trail(symbol, variable)
    else
      // two heap variables
      unify(symbol, variable)
  else
    return NULL

  return node
}
```

Figure 4.31: Pseudo-code for procedure `unify_with_variable`.

Now on the term stack we have `REF 2`, which dereferences to a structure on cell 4, and on node $n5$ we have the unbound trie variable `VAR0`, that gets bound to `STR 4` (Figure 4.32e).

Finally, the last term on the term stack is `b`, which can not be matched against `a` on node $n6$, hence no relevant answers are found on this trie.



(a) Time stamped trie and heap.

(b) At node $n1$.

(c) After unifying with node $n2$.

(d) After unifying with node $n4$.

(e) After unifying with node (e).

Figure 4.32: Unification of answer template {`REF 2,REF 2, b`} with time stamp 2.

4.5 Consuming Answers

Each consumer subgoal frame stores a linked list with all the answers collected during evaluation. This list is built incrementally and whenever a consumer choice point exhausts its answer list, the retrieval of new relevant answers is attempted in order to reflect the answers generated in the meantime by the generator subgoal.

While the retrieval of relevant answers is done in one step by searching the answer trie and pruning branches by time stamp and unification failure, the consumption of answers is done by consuming one answer at a time and is completely separated from the collection phase.

Consider an answer A with a trie path from a leaf node L to the trie root R. Consuming A amounts to unifying the symbols on the trie path to the answer template AT that was built for the consumer choice point. In such a way that, in the end, the variables on AT match the answer on the trie.

As we are certain that A unifies with AT, consumption is reduced to a simple unification between AT and the trie nodes from L to R. Implementation wise, we use the term stack that is initially pushed with AT and a symbol stack containing symbols from L to R. Then, we proceed by iteratively popping one term from the term stack and one symbol from the symbol stack and unifying one against the other.

In Figure 4.33, we present an example showing the data structures involved in consuming a subsumptive answer. The subsumptive subgoal is `p(X,Y,Z)` and the subsumed subgoal is `p(d,f(X),3)`. The answer to consume is `p(d,f(a),3)`, which corresponds to the binding {`X = a`}.

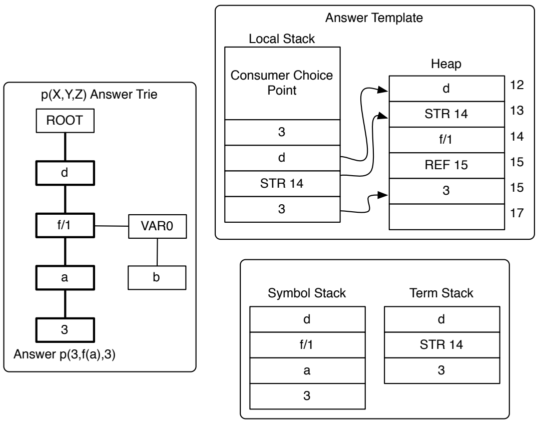


Figure 4.33: Data structures related to answer consumption.

4.6 Compiled Tries

After an answer trie is completed, we can optimize the process of consuming answers from a complete subgoal by annotating each trie node with a *trie instruction*. This optimization technique is called *compiled tries* [RRS+99].

Compiled tries are based on the observation that all common prefixes of the terms in a trie are shared during execution of trie instructions. Thus, when backtracking through the terms of a trie, each transition is taken at most only once.

In Figure 4.34, we represent a compiled answer trie for the subgoal `p(X,Y,Z)`. Notice that each node is extended with an instruction field. The instruction set follows the standard WAM instruction style with *try*, *retry* and *trust*. On a sibling chain the leftmost nodes are marked with *try* instructions and middle nodes with *retry* instructions. Rightmost nodes use *trust* instructions, while single nodes use *do* instructions.

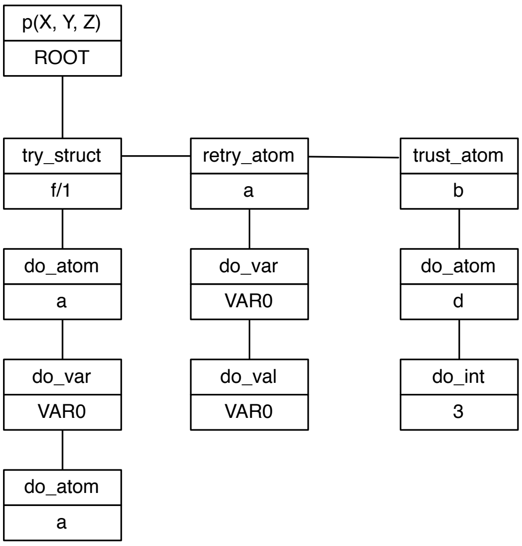


Figure 4.34: A compiled trie for subgoal `p(X,Y,Z)`.

A *try* instruction creates a new WAM choice point pointing to the sibling node, while *retry* instructions change the previous choice point to point to the next sibling node, thus enabling us to navigate to new trie branches while backtracking. The *trust* instruction removes the choice point as no more siblings are available. Finally, *do* instructions do not create choice points as no backtracking options are available.

In a variant tabling engine, each trie instruction just binds each variable on the substitution factor to a term. On structured terms, like functors or lists, the term is first built on the heap and then bound to the current variable, while the unbound arguments are then passed into the next trie levels to be instantiated.

In XSB, time stamped tries are used to evaluate subsumed subgoals while the subsuming subgoal is incomplete, thus providing incremental retrieving of answers. When a subgoal G completes, the engine uses the compiled answer trie from G to evaluate a subsumed subgoal G', instead of loading each individual answer. Not all answers of from G will be relevant to G', thus we must prune irrelevant paths through the process of unification.

Some modifications are thus required to use compiled tries in subsumptive tabling. First, instead of using the substitution factor when running compiled code, we use the answer template. Next, each instruction must now do unifications instead of simple bind operations, because in addition to unbound variables, these instructions can also receive instantiated terms. Finally, while each instruction succeeds when running with the substitution factor of the generator subgoal, for subsumed subgoals some instructions can fail, because not every answer will be relevant, hence will not unify with the answer template. The unification operations applied on each trie node are similar to the unifications done while collecting relevant answers in time stamped tries (Section 4.4).

Variant engines like YapTab collapse each hash table into a chain of nodes before compiling the trie. While this technique makes sense for a variant engine, in subsumptive engines having hash tables helps the unification process by enabling fast search of instantiated terms. If we had a very long sibling chain, locating a specific term would have a linear time complexity, while hash tables provide $O(1)$ search complexity. Thus, instead of removing hash tables, they must be kept on a subsumptive tabling engine. To use the hash tables, we must extend the trie instruction set with a new instruction, *do_hash* (Figure 4.35).

When the next term to unify is instantiated, we first lookup the bucket for this term and execute the code of the bucket chain. But periodically, we store a choice point that will execute code from the variable bucket, because variables can unify with any term.

If the term is a variable, we must visit every trie node in the hash table. This is done by storing a choice point that keeps the next hash bucket to be executed.

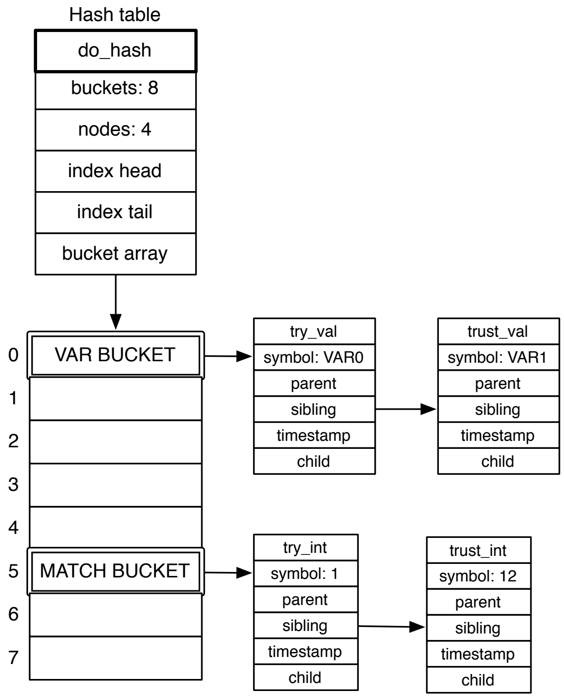


Figure 4.35: Compiled hash table.

4.7 Call Subsumption in YapTab

Our first attempt to extend the variant YapTab engine to support call by subsumption involved importing the subsumption related algorithms and data structures described in the previous sections from XSB into YapTab. These algorithms were imported as faithful as possible, hence very little modifications were made as C macros were used to translate the original code to YapTab. Sections of code specific to XSB or Yap were protected with conditional compilation, thus enabling both Prolog systems to use the same code.

In this section we will describe in more detail the modifications made to the following components of YapTab: tabled data structures, tabling operations and compiled tries. This new version of YapTab is called *YapTab_TST* and will be described in the next sections.

4.7.1 Data Structures

This section describes the modifications and extensions made to existent data structures in order to implement *YapTab_TST*.

4.7.1.1 Table Entry

Each table entry contains a bit field called **mode_flags** which stores information about the behavior of the corresponding tabled predicate. The original YapTab supports the following mutually exclusive flags: **batched** / **local**, for defining the scheduling strategy; and **load answers** / **exec answers**, the later makes the engine to use compiled tries, while the first forces the engine to load a completed table by loading answers individually.

Two new mutually exclusive flags were created for *YapTab_TST*: **variant**, which forces the predicate to use variant tabling and **subsumptive** to use subsumptive tabling.

4.7.1.2 Trie Nodes

YapTab uses two types of trie nodes: subgoal trie nodes, for subgoal tries; and answer trie nodes, for answer tries. In terms of hash tables, the same data structures are used for both subgoal and answer hash tables. *YapTab_TST* extends the answer and subgoal trie nodes with a **status** bit field and implements the time stamped trie nodes as answer trie nodes extended with a **timestamp** field. The **status** bit field is used to identify the type of trie node, i.e., if it is an hash table, a leaf node or an hashed node. For hash tables, we extended the answer hash table with the time stamp indexes to create the time stamped hash table. The index data structure was integrally copied from XSB.

4.7.1.3 Subgoal Frames

The subgoal frame structure in YapTab is a main component of the table space. It contains the fields: **answer_trie**, a pointer to the answer trie; **state**, the state flag; **first_answer** and **last_answer**, as the answer return list; **next**, a pointer to the next executing subgoal; and **generator_cp**, which points to the generator choice point.

Two new kinds of subgoal frame were created: the *subsumptive generator subgoal* *frame* and the *subsumed consumer subgoal frame.*

The subsumptive generator subgoal frame extends the original variant subgoal frame with the field **consumers**. This field points to a consumer subgoal frame and works as a chain link to the subsumed subgoal frames of the generator subgoal.

The subsumptive consumer subgoal frame does not extend the variant subgoal frame, because some variant fields are not needed. The consumer frame contains the following fields: **state**, the state of execution; **generator**, a pointer to the subsumptive generator subgoal frame; **consumer_cp**, a pointer to the first consumer choice point; **first_answer** and **last_answer**, as the answer return list; **ts**, the consumer time stamp; **next**, a pointer to the next evaluating consumer subgoal frame; **answer_template**, a pointer to the answer template built on the heap; and **consumers**, to link consumer subgoals of the generator subgoal frame.

Each subgoal frame was also extended with a **type** field, which identifies the subgoal frame type. The variant subgoal frame now uses an answer return list instead of using the **child** field of each trie node to link answers.

Figure 4.36 illustrates a subgoal trie for the predicate `p/2` with the following subgoals: `p(X,Y)`, as a generator subgoal; and `p(2,X)` and `p(3,X)` as consumer subgoals.

4.7.2 Tabled Subgoal Call

The tabled subgoal call operation is one of the four main tabling operations. In YapTab, if a subgoal call is already on the table space, a new consumer node is allocated and the execution runs the answer resolution operation, in order to consume answers. Otherwise, a new generator node is allocated and the subgoal compiled code is executed.

Figure 4.37 presents the pseudo-code for the original tabled subgoal call operation. In the consumer case, we use the `find_dependency_node` and `find_leader_node` procedures to locate the leader node for this consumer. The function `find_dependency_node` simply returns the choice point of the generator node for this subgoal, while `find_leader_node` iterates the dependency space between `dependency` and the new dependency frame for this consumer, to locate a dependency frame in which the leader is outside this range. If no such dependency frame exists, the leader node is set as the generator choice point found in `find_dependency_node`, else we use the leader of the dependency frame that satisfied the previous condition. [Roc01]

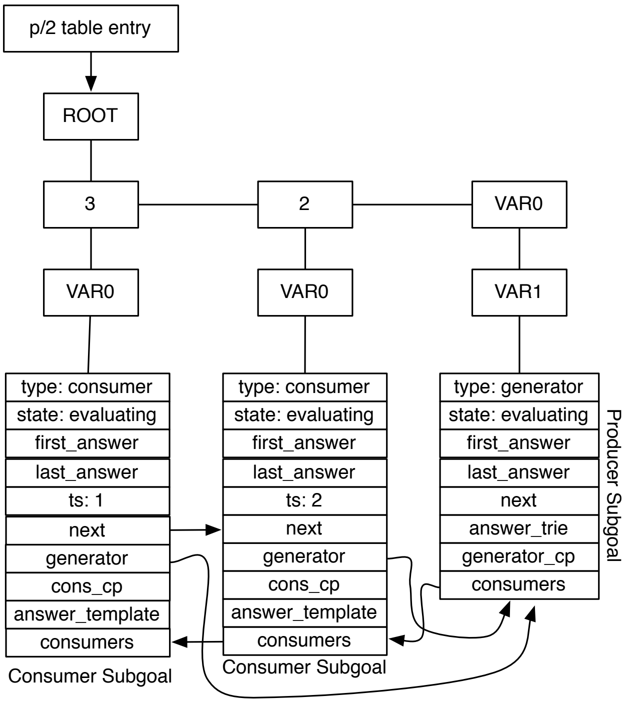


Figure 4.36: Subgoal trie with a subsumptive generator and two subsumed consumer subgoal frames.

```
tabled_subgoal_call(table_entry, arguments) {
  subgoal_frame = subgoal_search(subgoal_trie(table_entry), arguments)

  if (state(subgoal_frame) == READY)
    // new generator
    store_generator_node(table_entry, arguments, subgoal_frame)
    jump_to_predicate_code()
  else if (state(subgoal_frame) == EVALUATING)
    // new consumer
    dependency = find_dependency_node(subgoal_frame)
    leader = find_leader_node(subgoal_frame, dependency)
    store_consumer_node(table_entry, subgoal_frame, leader)
    jump_to_answer_resolution()
  else
    // subgoal is completed
    if (state(answer_trie(subgoal_frame)) != COMPILED)
      compile_answer_trie(answer_trie(subgoal_frame))
    execute_answer_trie(answer_trie(subgoal_frame))
}
```

Figure 4.37: Pseudo-code for the original tabled subgoal call operation.

YapTab_TST extends the tabled subgoal call operation to deal with either subsumptive and variant subgoals, and if a predicate uses call by subsumption, we call the `subsumptive_subgoal_search` procedure (Figure 4.38) to look on the subgoal trie for subsuming subgoals. If no subsuming path is found, we insert the subgoal path on the subgoal trie and create a new generator subgoal frame. If some path is found and the subgoal frame is a generator it can be either a variant of our subgoal or a more general subgoal. For both cases we use `construct_answer_template_from_lookup`, which constructs the right answer template for us. If the found subgoal frame is a consumer, we must consume from its generator and reconstruct the answer template by using the generator trie path. Consumer trie paths are only constructed when the generator subgoal frame is still evaluating. When the generator is completed, we just execute the compiled trie, so there is no need to create a consumer subgoal frame.

```
subsumptive_subgoal_search(table_entry, arguments) {
  (path, leaf) = lookup_subsuming_call(subgoal_trie(table_entry), arguments)

  if (path == NO_PATH)
    leaf = insert_with_variant_continuation(subgoal_trie)
    local_stack = construct_answer_template_from_insertion(local_stack)
    return create_generator_subgoal_frame(leaf)
  else
    found_sf = subgoal_frame(leaf)

    if (type(found_sf) == SUBSUMPTIVE_GENERATOR)
      subsumer_sf = found_sf
      local_stack = construct_answer_template_from_lookup(local_stack)
    else
      subsumer_sf = generator(found_sf)
      local_stack = construct_answer_template_from_generator(subsumer_sf, local_stack)

    if (path_type == VARIANT_PATH)
      return found_sf

    // subsumptive path
    if (state(subsumer_sf) == EVALUATING)
      // create variant path
      leaf = insert_with_variant_continuation(subgoal_trie)
      copy_answer_template_to_heap(local_stack)
      return create_consumer_subgoal_frame(leaf, subsumer_sf)
    else
      return subsumer_sf
}
```

Figure 4.38: Pseudo-code for procedure `subsumptive_subgoal_search`.

When a consumer subgoal frame is created we make a *structural copy* of the answer template into the heap. A structural copy is made by recursively copying structured terms (functors and lists) and by making integral copies of constant terms. For variables, we create a new variable on the heap for each variable on the local stack, in such a way that no references exist between the two answer templates. Figure 4.39

illustrates a structural copy of the answer template {`X, f(a)`} between the local stack and the heap.

The answer template on the heap will be used as an argument to the algorithm that collects relevant answers from the generator answer trie. Each consumer subgoal frame has just one copy of the answer template on the heap, while each consumer node uses its own answer template built on the local stack to consume answers, because it references variables and terms from the arguments.

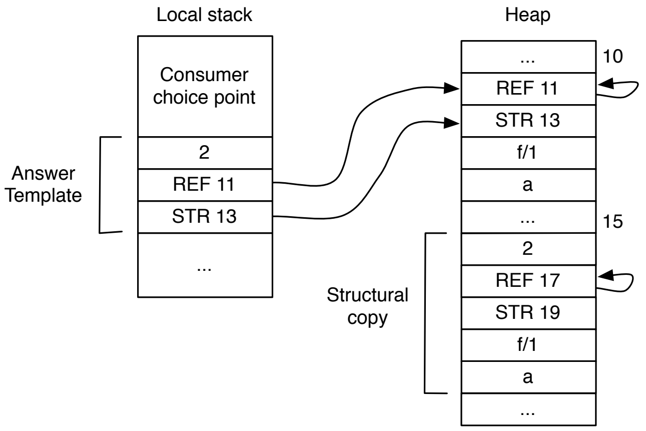


Figure 4.39: Making a structural copy of the answer template {`X, f(a)`}.

The field **consumer_cp** of the consumer subgoal frame is set to point to the first call of the consumer subgoal and will be used to access the value of `H` stored on the choice point. This value of `H` points to the top of the heap during the choice point creation which corresponds to the answer template copied before. The field **answer_template** points to `H` and is used to avoid accessing the choice point, thus it must be recalculated during garbage collection.

This differs from XSB [Joh00] where a *non-structural copy* of the answer template is built on the heap for each consumer node. A non-structural copy is made by making references from the heap to the terms on the heap. Figure 4.40 illustrates a non-structural copy of the answer template {`X, f(a)`}. while making non-structural copies is faster than doing structural copies, collecting relevant answers can, potentially, involve more environment switching, because we must reconstitute the environment of the consumer to ensure that the answer template is valid, which involves using the trail to unbind and/or rebind variables. Our approach has the disadvantage of making structural copies, but only one copy is done and there is no need to invoke the algorithm within the environment of the subsumed call.

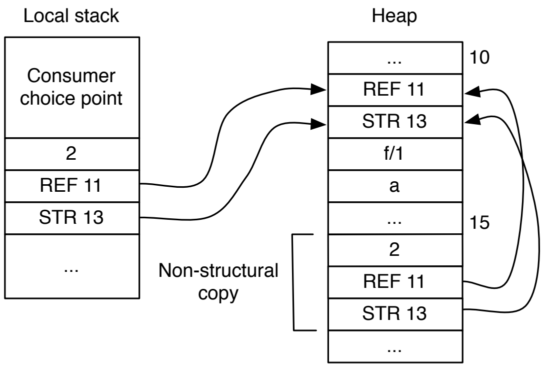


Figure 4.40: Making a non-structural copy of the answer template {X, f(a)}.

Other modifications were applied to the tabled call operation in order to abstract some details about variant and subsumptive tabling. Figure 4.41 shows the updated tabled subgoal operation.

```
tabled_subgoal_call(table_entry, arguments) {
  subgoal_trie = subgoal_trie(table_entry)

  if(method(table_entry) == VARIANT)
    subgoal_frame = variant_subgoal_search(subgoal_trie, arguments)
  else
    subgoal_frame = subsumptive_subgoal_search(subgoal_trie, arguments)

  if (is_new_generator_call(subgoal_frame)) // CHANGED
    store_generator_node(table_entry, arguments, subgoal_frame)
    jump_to_predicate_code()
  else if (is_new_consumer_call(subgoal_frame)) // CHANGED
    dependency = find_dependency_node(subgoal_frame)
    leader = find_leader_node(subgoal_frame, dependency)
    store_consumer_node(table_entry, subgoal_frame, leader)
    // NEW
    if ((type(subgoal_frame) == SUBSUMED_CONSUMER) and (state(subgoal_frame) == READY))
      recompute_answer_template(subgoal_frame)
      consumer_cp(subgoal_frame) = B
      add_to_consumer_stack(subgoal_frame)
    jump_to_answer_resolution()
  else
    // subgoal is completed
    if (state(answer_trie(subgoal_frame)) != COMPILED)
      compile_answer_trie(answer_trie(subgoal_frame))
    execute_answer_trie(answer_trie(subgoal_frame))
}
```

Figure 4.41: Pseudo-code for the new tabled subgoal call operation.

The function `is_new_generator_call` (Figure 4.42) inspects the found subgoal frame in order to tell if a new generator node must be allocated. For call by subsumption, a new generator is allocated whenever a variant subgoal is not found on the trie and the resulting subgoal frame is a generator.

```
is_new_generator_call(subgoal_frame) {
  if (type(subgoal_frame) == VARIANT or type(subgoal_frame) == SUBSUMPTIVE_GENERATOR)
    // must be a first call to either a variant or subsumptive generator subgoal
    return state(subgoal_frame) == READY

  // consumer subgoal are not generator calls
  return FALSE
}
```

Figure 4.42: Pseudo-code for function `is_new_generator_call`.

The function `is_new_consumer_call` (Figure 4.43) decides that a consumer node must be allocated when:

1. The subgoal frame is variant or generator and is currently being evaluated;

2. Or, the subgoal frame is consumer and the generator subgoal is still evaluating.

```
is_new_consumer_call(subgoal_frame) {
  if (type(subgoal_frame) == VARIANT or type(subgoal_frame) == SUBSUMPTIVE_GENERATOR)
    // equal to the old tabled subgoal operation
    // but also considering subsumptive generator calls
    return state(subgoal_frame) == READY

  // consumer subgoal frames
  return state(generator(subgoal_frame)) == EVALUATING
}
```

Figure 4.43: Pseudo-code for function `is_new_consumer_call`.

Another important change involves calculating the leader node of a subsumptive consumer node. Surprisingly, only the algorithm to find the dependency node must be altered, as the dependency node of a subsumed consumer is the generator choice point of the generator subgoal frame (Figure 4.44).

If the subsumed subgoal is called for the first time, we copy the answer template to the heap and the `consumer_cp` is made to point to the current choice point. The subgoal frame is also pushed into the consumer subgoal frame stack in order to be completed in the completion operation (see next section for more details).

```
find_dependency_node(subgoal_frame) {
  if (type(subgoal_frame) == VARIANT or type(subgoal_frame) == SUBSUMPTIVE_GENERATOR)
    return generator_cp(subgoal_frame)
  else
    return generator_cp(generator(subgoal_frame))
}
```

Figure 4.44: Pseudo-code for the new `find_dependency_node` procedure.

4.7.3 Answer Resolution and Completion

The tabling operations answer resolution and completion both check if a given consumer node has unconsumed answers. The completion operation iterates over the dependency space to look for consumer nodes with unconsumed answers and the answer resolution operation attempts to consume the next answer of a consumer and also iterates over the dependency space in order to run the completion algorithm.

In order to abstract away if a consumer node has answers to consume, we created a function that distinguishes between both variant and subsumptive cases (Figure 4.45). The function accepts a dependency frame (note that there is a dependency frame for each consumer node) and returns an answer continuation. An answer continuation is represented by a pointer to a linked list with two fields: **answer**, the answer itself; and **next**, which points to the next element of the list, if any.

With subsumptive consumer nodes we first verify if the answer return list of the consumer subgoal frame contains more answers from the saved continuation. If there is any, we return the next answer from this list and consume it. If the list has no more answers, we inspect if the consumer time stamp is equal to the generator time stamp, which means that no new answers were generated for the general subgoal. In the other hand, if new answers are available, we run `tst_collect_relevant_answers` to collect any relevant answers for this consumer that can be appended into the answer return list and returned for consumption. In any case, the time stamp of the consumer is thus updated to avoid collecting repeated answers in future iterations. Remember that once a single consumer node collects newer answers, every consumer node of a subsumed subgoal will see them, thus enabling sharing of answers among the consumer nodes. Note that in order to run this algorithm, the dependency frame data structure was extended with a new **sg_fr** field.

Figure 4.46 presents the new completion operation, using the devised abstractions.

Once a generator subgoal completes we must mark each subgoal that appears under

```
get_next_answer_continuation(dependency_frame) {
  sg_fr = sg_fr(dependency_frame)
  last_continuation = last(dependency_frame)
  next_continuation = next(last_continuation)

  if (type(sg_fr) == VARIANT or type(sg_fr) == PRODUCER)
    return next_continuation

  // subsumed consumer subgoal frame case

  if (next_continuation != NULL)
    // no need to collect answers from the generator's answer trie
    // as answers are still available on the answer return list
    return next_continuation

  // must collect new available answers, if any
  consumer_ts = timestamp(sg_fr)
  generator = generator(sg_fr)
  generator_ts = timestamp(answer_trie(generator))

  if (generator_ts == consumer_ts)
    return NULL

  // collect answers
  answer_list = tst_collect_relevant_answers(answer_trie(generator),
        consumer_ts, answer_template(sg_fr))
  timestamp(sg_fr) = generator_ts

  if (answer_list != NULL)
    append_return_list(sg_fr, answer_list)

  return answer_list
}
```

Figure 4.45: Pseudo-code for function `get_next_answer_continuation`.

the SCC as *complete*. Hence, some modifications must be made to complete subsumed consumer subgoal frames.

In YapTab each variant subgoal frame is stacked into the *subgoal frame stack* during execution. The top of the stack can be accessed by using `TOP_SG_FR`. On completion, the subgoal frame stack is iterated until the leader subgoal frame is reached. We used this stack to complete for both variant and subsumptive generator subgoal frames.

Subsumed consumer subgoal frames are pushed into a stack called *consumer subgoal frame stack*. During completion, the subgoal frames with a consumer choice point (`consumer_cp`) younger than the completion point are completed. Figure 4.47 illustrates the subgoal frame stack, the consumer subgoal frame stack and the dependency space.

```
completion(generator) {
  if (is_leader_node(generator))
    df = TOP_DEP_FR
    while (younger_than(consumer_cp(df), generator))
      cont = get_next_answer_continuation(dep_fr) // CHANGED
      if (cont)
        // unconsumed answers
        back_cp(df) = generator
        consumer = consumer_cp(df)
        restore_bindings(CP_TR(generator), CP_TR(consumer))
        goto answer_resolution(consumer)
      df = next(df)
    perform_completion()
    adjust_freeze_registers()
  backtrack_to(CP_B(generator))
}
```

Figure 4.46: Pseudo-code for the new completion operation.

Some data structures, like the time stamp indexes, are deleted during completion from the subsumptive generator subgoal frames.

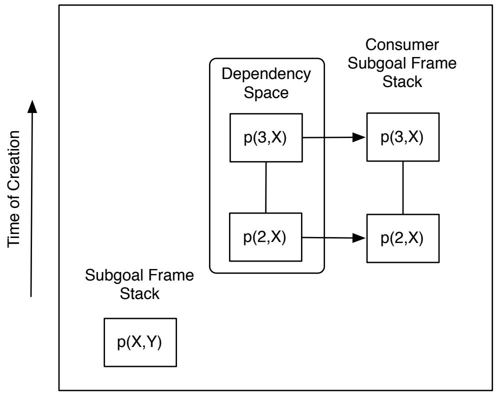


Figure 4.47: Subsumptive generator and consumers before completion.

4.7.4 Compiled Tries

YapTab also implements the compiled tries optimization, but as described in Section 4.6 we needed to change each trie instruction to do unification instead of simple variable binding.

Another important modification was the hash instructions. Two new instructions were implemented: `trie_do_hash` and `trie_retry_hash`. The instruction `trie_do_hash` is executed once an hash table node is reached. If the next term to unify is a variable (case 1) each hash bucket will be executed, if it is an instantiated term (case 2) the indexed bucket will be executed first, followed by the variable bucket.

We use an hash choice point (`hash_choice_pt`) to execute each bucket. For instantiated terms, the hash choice point is only allocated when both variable and indexed bucket exist and are different. For variables, the choice point is always stored. The instruction set to execute upon backtracking is `trie_retry_hash`.

An hash choice point is an WAM choice point extended with two new fields: **last_bucket** and **final_bucket**. The field **last_bucket** points to the last executed bucket on case 1 or the variable bucket for case 2. The second field, **final_bucket**, helps differentiate between case 1 and 2, as in case 2 its value is NULL. In case 1 points to the final hash table bucket, hence it is easy to know when to remove the choice point.

4.8 Chapter Summary

This chapter throughly explained the algorithms and data structures behind the Time Stamped Tries technique to implement call by subsumption in a tabling engine. We discussed how the answer trie was extended with time stamp information and explained the need for a time stamp index to efficiently collect relevant answers for subsumed subgoals.

We discussed how the YapTab engine was extended with the Time Stamped Tries mechanisms to provide tabling by call subsumption. We also showed how the core algorithms in YapTab were minimally affected.

In the next chapter, we will present a new tabling extension called Retroactive Call Subsumption (RCS), that improves upon call subsumption by enabling bidirectional sharing of answers, that is, answers will be shared even if a subsumed subgoal is called before a subsuming subgoal.

Chapter 5

Retroactive Call Subsumption

This chapter explores the concept of *Retroactive Call Subsumption* (RCS). RCS enables full sharing of answers among subsumptive subgoal calls, independently of the order they are called.

We start by introducing the motivation behind RCS by illustrating the shortcomings of traditional call subsumption mechanisms. Next, we present the concepts introduced by RCS and how execution rules are extended to support retroactive-based tabling. Other extensions are then discussed, namely: the new table space organization based around the ideas of the *common global trie* proposal [CR08] and the algorithm to traverse the subgoal trie to search for subsumed subgoals. Finally, we give some details about the implementation of this new extension in the YapTab system.

5.1 Motivation

In traditional call subsumption, a new call to the subgoal G is considered a generator subgoal when it is called for the first time and a more general subgoal G' is not found on the subgoal trie. When G' exists, G is considered a consumer subgoal and a new consumer node is allocated to consume answers from the subsuming subgoal G'.

Consider two subgoals, `p(X,1,2)` and `p(X,1,Z)` , and that subgoal `p(X,1,2)` is called first, followed by `p(X,1,Z)`. When `p(X,1,2)` is called, it is considered a generator subgoal as no subgoals exist on the subgoal trie. The subgoal `p(X,1,Z)` is also considered a generator, because `p(X,1,2)` does not subsume `p(X,1,Z)`. However, If the call order is swapped, `p(X,1,Z)` is still considered a generator subgoal, but now `p(X,1,2)` finds `p(X,1,Z)` as a subsuming subgoal on the subgoal trie, and thus it is considered a consumer subgoal.

While call subsumption provides good results in terms of memory usage and execution time, it suffers from a major problem: the order in which the subgoals are called can greatly affect the performance and applicability of the technique. To solve this problem, we introduce a new mechanism, called *Retroactive Call Subsumption* (RCS), that retroactively modifies active tabled nodes in order to enable full sharing of answers between subsuming and subsumed subgoals, independently of the order they are called. Please notice that retroactive-based tabling is only applicable if using a batched scheduling strategy.

5.2 General Idea

The key idea of Retroactive Call Subsumption is to stop the computation of the subsumed subgoals that are currently running, by transforming those generator subgoals into consumer subgoals. Thus, instead of generating their own answers by means of code execution, they will consume answers from the more general subgoal that has been called.

When a generator subgoal G executes, an arbitrary number of choice points directly related to G can be created to compute G's answers. Therefore, when G is to be transformed into a consumer subgoal, we must *selectively prune* the parts of the computation that are related to G and transform G's generator choice point in such a way that it will consume answers from the new generator subgoal, instead of generating its own answers. Pruning the computation of G can thus potentially save execution time as G no longer properly executes but consumes answers from the more general subgoal.

Consider the program in Figure 5.1 that uses RCS and the query goal '`a(X), p(Y,Z)`'. The goal `a(X)` starts by calling `p(1,X)`, which succeeds with the answer {`X = 3`}. By following forward execution, `p(Y,Z)` is called in the continuation and it then verifies if any subsumed subgoal is currently running (Figure 5.2 (a)). It finds `p(1,X)` and thus it marks this subgoal frame as a consumer subgoal frame that will consume from `p(Y,Z)`. In order for `p(1,X)` to act as a consumer, its generator choice point is transformed into a *retroactive choice point*, which amounts to update the *continuation alternative* (**CP_AP** choice point field) to an instruction called *retroactive_resolution*, which implements the needed mechanisms to control the evaluation of a *retroactive node* (Figure 5.2 (b)).

```
:- use_retroactive_tabling p/2.

a(X) :- p(1, X).

p(1, 3).
p(2, 3).
p(1, 2).
```

Figure 5.1: An example of a program using retroactive tabling.

Next, `p(Y,Z)` continues execution and a new answer is generated, {`Y = 1, Z = 3`}. By means of backtracking, all answers for `p(Y,Z)` are generated and the subgoal completes. Execution then returns to the retroactive choice point of `p(1,X)` and retroactive resolution is employed. As the generator subgoal `p(Y,Z)` has already completed, `p(1,X)` can be turned into a loader node in order to consume all the new matching answers found by `p(Y,Z)`. In this case, the only matching answer is {`X = 2`} (Figure 5.2 (c)). Note that a retroactive node can be transformed into other types of nodes, as it will become clear in the next sections.

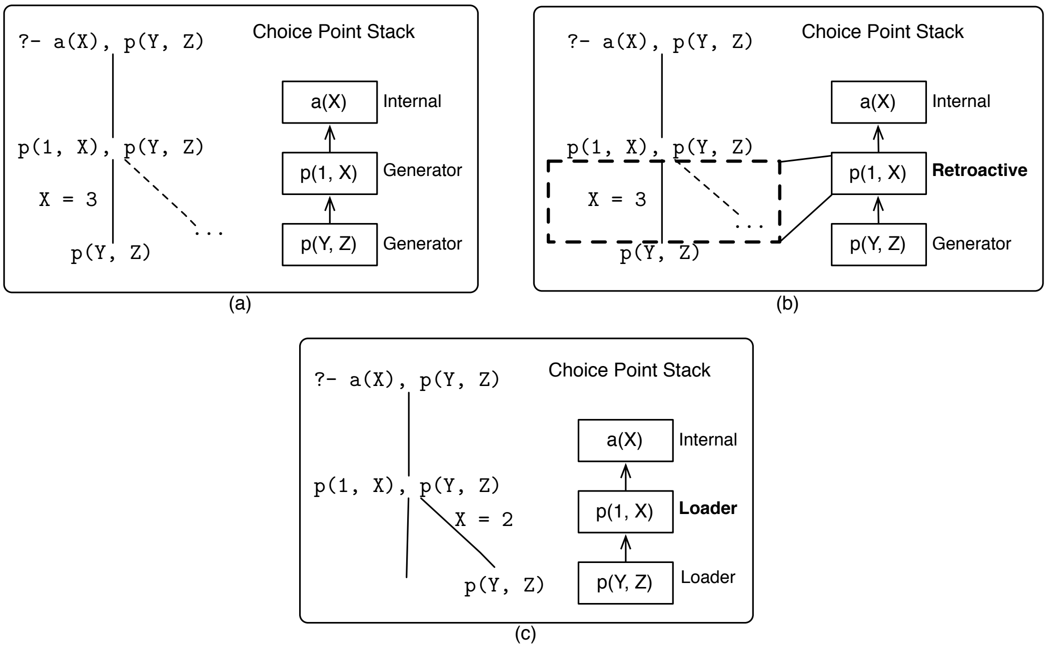


Figure 5.2: Evaluating '`a(X), p(Y,Z)`' using retroactive tabling.

The previous example has illustrated one special type of pruning called *external pruning*. External pruning occurs when the subsuming subgoal G' is an *external* *subgoal* to the evaluation of the subsumed subgoal G. Another type of pruning is called *internal pruning* and happens when G' is an *internal subgoal* to the evaluation of G, that is, G' is called as a part of the evaluation of G. Although these two basic types of pruning can derive any other situation, they both must deal with the same issues related to pruning. These issues will be explored in detail in the next section.

5.3 Retroactive Pruning

When pruning parts of the computation, we must know the areas of the local stack that contain the choice points to prune. Given the nature of tabled evaluation, choice points not directly related to the pruned subgoal can get mixed with other choice points. This happens when a branch containing external choice points has been suspended, but after backtracking to internal choice points we execute a subsuming subgoal. Therefore, we must have a mechanism that can tell us if a certain choice point is internal to a subgoal and a mechanism that computes the range of choice points from which to potentially prune.

5.3.1 Subgoal Dependency Tree

To solve the problem of determining if a tabled node is internal to a subgoal, we construct a *subgoal dependency tree* by extending the subgoal frame and the dependency frame data structures with a field called **top_gen**. This field is a pointer to the top generator subgoal of the corresponding tabled node. Hence, we can use the subgoal dependency tree to know if the subgoal A is internal to the subgoal B. For this, we traverse the `top_gen` links until: (1) subgoal B is reached, thus A is internal to B; we reach an older subgoal than B, therefore A is not internal to B. We use B's generator choice point as the threshold for abandoning search and declaring A external of B. Figure 5.3 presents the pseudo-code for the `is_internal_subgoal_frame` function.

The **top_gen** field is initialized with the value of a global variable called `TOP_GEN`. This variable is used in various situations: (1) when a tabled subgoal call creates a generator node we set the new subgoal frame's **top_gen** field to `TOP_GEN` and then update `TOP_GEN` to the new subgoal frame; (2) when the new answer tabled operation is executed we set `TOP_GEN` to the value of the subgoal frame's **top_gen** field; (3) when a consumer node is created (and the respective dependency frame) we use the `TOP_GEN` value to set the dependency frame's **top_gen** field; (4) finally, when a consumer node

```
is_internal_subgoal_frame(target_fr, subgoal_fr, min) {
  if (target_fr == subgoal_fr)
    return TRUE

  sg_fr = top_gen(subgoal_fr)

  while (sg_fr && younger_or_equal(generator_cp(sg_fr), min))
    if (sg_fr == target_fr)
      return TRUE

    sg_fr = top_gen(sg_fr)

  return FALSE
}
```

Figure 5.3: Pseudo-code for function `is_internal_subgoal_frame`.

is resumed we set `TOP_GEN` to the value of the dependency frame's **top_gen** field.

5.3.2 Computing Stack Limits

One important input parameter of pruning are the choice points relevant to the computation of target subgoal. These parameters are known as `min_cp` and `max_cp` and represent the generator choice point of the subsumed subgoal and the bottom-most choice point involved in its computation, respectively. These values are computed by first storing the address of the generator choice point as the upper limit, and the lower limit is updated to the lowest value of **B_FZ** or **B** when: (1) a new answer is generated, (2) a new clause is executed, or (3) completion is attempted. While this is a simple approach, some parts of the local stack can belong to an *external generator*. Updating the bottom limit to **B_FZ** allow us to cover areas of the stack that were frozen during evaluation of the subgoal.

5.3.3 Basic Issues

When pruning execution branches, issues arise mostly when the computation of the subsumed subgoal involves consumer and/or generator nodes. As an example, consider the program in Figure 5.4 that mixes retroactive and variant-based tabled predicates. For this program, we will use the query goal '`a(X,Y), p(Z,W)`'.

Initially, the evaluation calls `a(X,Y)` and a new generator node is stored. Next, the retroactive subgoal `p(1,X)` is called and because no subsuming subgoal is found, a new

```
:- use_variant_tabling [a/2, b/1].
:- use_retroactive_tabling p/2.

a(X, Y) :- p(1, X), b(Y).
a(3, 4).

b(1).
b(2).

p(1, X) :- a(_, X).
p(1, X) :- b(X).
```

Figure 5.4: An example of a program using retroactive tabling and variant tabling.

generator node is created. The first clause of p/2 then executes a(_,X), which is a consumer of a(X,Y), but, as no answers are available to consume, execution suspends this node and backtracks to try the second clause of p/2. Here, b(X) is called for the first time, creating a new generator choice point. An answer for b(X) is then found, {X = 1}, and by forward execution it is also an answer for p(1,X). In the continuation, b(Y) is called, creating a new consumer node that consumes the answer {X = 1} and by forward execution, a first answer for a(X,Y), {X = 1, Y = 1}, is generated.

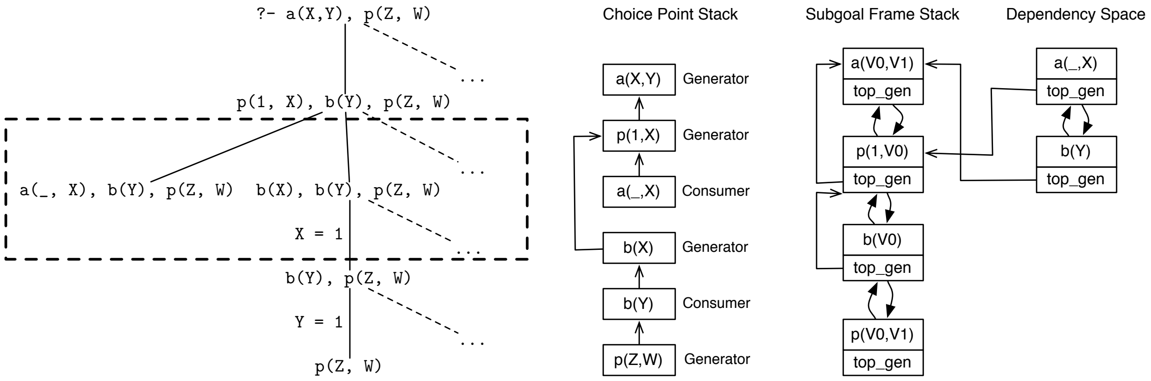


Figure 5.5: Evaluating ‘a(X,Y), p(Z,W)’ using retroactive tabling.

Next, subgoal p(Z,W) is called, which subsumes p(1,X) and thus the evaluation of this subgoal must be pruned (Figure 5.5). As p(Z,W) is external to p(1,X), we have a case of external pruning. Please notice that p(1,X) includes two choice points involved in its computation, namely a(_,X) and b(X), which must be pruned.

The consumer node associated with subgoal a(_,X) must leave the computation and for that its dependency frame is removed from the dependency space.

Pruning the generator node associated with the subgoal b(X) is a more tricky case.

Notice that the consumer b(Y) depends on this subgoal to consume new answers, thus by removing the generator node the consumer will become an *orphaned consumer* and the computation will not complete properly. Therefore, we mark the subgoal b(X) as *pruned* and turn the consumer node of b(Y) into a retroactive node. Finally, the *previous choice point* field of the choice point associated with b(Y) must now point to p(1,X), because we must prevent the evaluation to step into pruned branches by means of backtracking. The choice point of b(Y) is called a *frontier choice point.* Figure 5.6 shows the state of the computation after pruning.

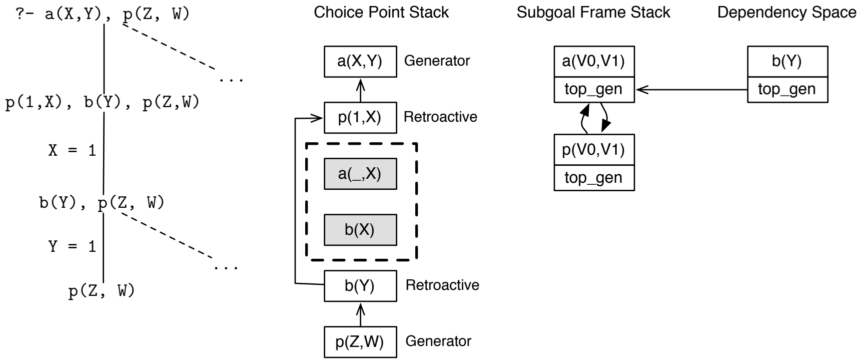


Figure 5.6: Executing external pruning over choice points belonging to the subsumed subgoal.

Next, the subgoal p(Z,W) starts to execute the first clause of p/2 and creates a new consumer for a(_,X), that will consume the answer found previously. By forward execution, an answer for p(Z,W), {Z = 1, W = 1}, is generated. By means of backtracking, the second clause of p/2 is executed and b(W) is called. As the subgoal b(VAR0) is a pruned subgoal, we first load the answers already generated for this subgoal and then execute the clauses of b/1. After b(W) generates all the answers, it completes successfully and execution backtracks to p(Z,W), to attempt completion of this subgoal. As there is a younger consumer node, a(_,W), that depends on an older subgoal, the completion operation cannot be done because p(Z,W) is not the leader.

We backtrack to the retroactive node b(Y) in order to do retroactive resolution. Evaluation notices that the subgoal has already completed, thus the retroactive node is transformed into a loader node to consume all answers that were not consumed yet. By forward execution, a new consumer for p(Z,W) is created that will generate further answers for the query goal. Once b(Y) does not have more answers to consume, execution backtracks to the retroactive node of p(1,X). Here, we note that the generator subgoal, p(Z,W) has still not completed and this retroactive node must be turned into a consumer node, which amounts to create a new dependency frame that is added into the dependency space, in order to participate in the resolution process.

After `p(1,X)` executes the answer resolution operation, evaluation backtracks to `a(X,Y)` that will execute the second clause of `a/2`. Once `a(X,Y)` executes the completion operation and each each consumer has consumed its answers, the subgoal completes and evaluation is finished.

5.3.4 Pruning Actions

The previous example has illustrated some of the different actions that must be applied to the choice points that belonging to the computation of a pruned subsumed subgoal. These actions dependent on the choice point type and are summarized in the next subsections.

5.3.4.1 Interior Nodes

Interior nodes are related to normal Prolog execution and can be easily pruned by ignoring them altogether. This approach, while simple, suffers from the problem of *trapped choice points*. This problem also affects the normal execution of delaying based tabling engines like YapTab and SLG-WAM, where choice points under consumers are frozen and remain until completion. The CHAT approach to tabling solves this problem by removing trapped choice points [DS99]. Another solution would involve modifications to the WAM garbage collector to collect unused space on the choice point stack.

5.3.4.2 Internal Consumers

Internal consumers must be explicitly pruned by removing the associated dependency frame from the dependency space. This prevents the resolution process to reactivate pruned branches. In the previous example, `a(_,X)` was an internal consumer.

Figure 5.7 shows the procedure `abolish_dependency_frames` that abolishes internal dependency frames given the subsumed subgoal to prune, `subsumed_sg`; the first choice point address from the range of choice points to selectively prune, `min_cp`; and the last choice point from the pruned range, `max_cp`. First, we ignore dependency frames younger than the `max_cp` choice point. Next, we iterate the dependency frames

```
abolish_dependency_frames(subsumed_sg, min_cp, max_cp) {
   dep_fr = TOP_DEP_FR

   while (dep_fr != NULL and younger_than(cons_cp(dep_fr), max_cp))
     top = previous(dep_fr)

   while (top != NULL and younger_than(consumer_cp(dep_fr), min_cp))
      previous_dep_fr = previous(dep_fr)

      if (is_internal_dependency_frame(subsumed_sg, dep_fr, min_cp))
         remove_dependency_frame_from_stack(dep_fr)
         free(dep_fr)

      dep_fr = previous_dep_fr
}
```

Figure 5.7: Pseudo-code for procedure `abolish_dependency_frames`.

inside the range and check if they are internal to the choice point by using the `is_internal_dependency_frame` function. This function uses the `top_gen` field of the dependency frame to traverse the chain of subgoal frames in order to reach the `subsumed_sg` subgoal frame. If we reach it, we remove the dependency frame from the dependency space. If the `subsumed_sg` subgoal cannot be reached within the limits of `min_cp`, the dependency frame is not internal and thus it is not pruned from the computation.

5.3.4.3 Internal Generators

For internal generators we must remove its corresponding subgoal frame from the subgoal frame stack and alter its state to *pruned*. Generally, when pruning internal generators, we have two situations: (1) the generator does not have consumers that are external to the computation of the subsumed subgoal; or (2) the generator has external consumers. The former situation does not introduce any problem, but the latter origins orphaned consumers. In our previous example, the consumer node for `b(Y)` is an orphaned consumer.

Usually, a pruned generator is called again during the evaluation of the subsuming subgoal, and before the computation reaches any of the orphaned consumers. Once reactivated, the subgoal frame for the pruned generator is pushed again into the top of the subgoal frame stack and its state is altered to *evaluating*. Then, the new generator node starts by consuming the previously generated answers and only then executes the program clauses.

```
abolish_subgoal_frames(subsumed_sg, min_cp, max_cp) {
   sg_fr = next(subsumed_sg)

   remove_subgoal_frame_from_stack(subsumed_sg)

   while (sg_fr != NULL and !younger_than(generator_cp(sg_fr), max_cp))
      next_sg_fr = next(sg_fr)

      if (is_internal_subgoal_frame(subsumed_sg, sg_fr, min_cp))
         sg_cp = generator_cp(sg_fr)

         remove_subgoal_frame_from_stack(sg_fr)

         if (type(sg_fr) == VARIANT or type(sg_fr) == RETROACTIVE)
            if (has_external_consumers(sg_fr))
               update_external_consumers(specific_sg, sg_fr, sg_cp, max)
               state(sg_fr) = pruned
            else
               free(sg_fr)
         else if (type(sg_fr) == SUBSUMPTIVE)
            if (has_external_subsumed_consumers(sg_fr))
               transform_external_subsumed_consumers(specific_sg, sg_fr, sg_cp, max,
                  has_external_variant_consumers(sg_fr))

            if (has_external_variant_consumers(sg_fr))
               update_external_consumers(specific_sg, sg_fr, sg_cp, max)
               state(sg_fr) = pruned
            else
               delete_from_subgoal_trie(sg_fr)
               free(sg_fr)

      sg_fr = next_sg_fr
}
```

Figure 5.8: Pseudo-code for procedure `abolish_subgoal_frames`.

Figure 5.8 shows the procedure `abolish_subgoal_frames` that is responsible to abolish internal generators. For variant and retroactive-based tabled subgoals we remove the subgoal frame from the stack (using `remove_subgoal_frame_from_stack`) and then we check if the subgoal has external consumers. If those consumers are found, they are turned into retroactive nodes by the procedure `update_external_consumers`.

Our system is also able to mix tabled subgoals using traditional call subsumption with retroactive-based tabling. For this case we distinguish between variant consumers (identical consumers by variable renaming) and subsumed consumers (proper subsumed consumers). Both consumer nodes are also transformed into retroactive nodes.

If a subgoal G has no external variant consumers, we remove G's subgoal frame from the system and its path from the subgoal trie, which means that external subsumed consumers of G are also turned into generators as the generator G has been removed. The fifth argument of the procedure `transform_external_subsumed_consumers` indicates whether external variant consumers exist and for that we use the function `has_external_variant_consumers`. When we have external variant consumers, the pruned subgoal G changes its state to *pruned* in order to be reactivated later by a new call to it or by an orphaned consumer, and the external subsumed consumers remain unaltered.

A tricky situation happens when an orphaned variant consumer of G is later pruned by another subgoal and the external subsumed consumers of G are left in a situation where the generator subgoal G will not be reactivated. In this case, we check for situations where G has no more variant consumers and we change its state to *dead*. Later on, when an external subsumed consumer is reactivated by means of retroactive resolution, we verify if the generator G is *dead*. If this is the case, we simply convert the subgoal frame to a subsuming subgoal frame and turn the retroactive node into a generator. This also involves modifications to the answer template, which must be transformed into a generator answer template (with only variables).

For an example using subsumptive subgoals, consider the program in Figure 5.9 and the query goal '`p(1,A), t(1,2,B), b(1,C), p(D,E), b(F,G)`'.

```
:- use_retroactive_tabling [b/2, p/2].
:- use_subsumptive_tabling t/3.

p(X, 55) :- t(X, A, B).
p(1, 5).
p(10, 10).

b(X, 20) :- t(X, A, B).
b(3, 1).

t(1, 2, 3).
t(1, 2, 5).
t(3, 10, 20).
```

Figure 5.9: An example of a program using retroactive tabling and subsumptive tabling.

The first goal `p(1,A)` creates a new generator node and calls `t(1,A,B)`, which is considered a subsumptive generator subgoal. By forward execution we then call the subgoal `t(1,2,B)`, which will create a consumer node that subsumes the subgoal `t(1,A,B)`. In the continuation, we call the retroactive generator subgoal `b(1,C)` which calls t(1,A,B), a variant consumer of the initial subsumptive generator. Next, we call p(D,E) and p(1,A) must be pruned (Figure 5.10).

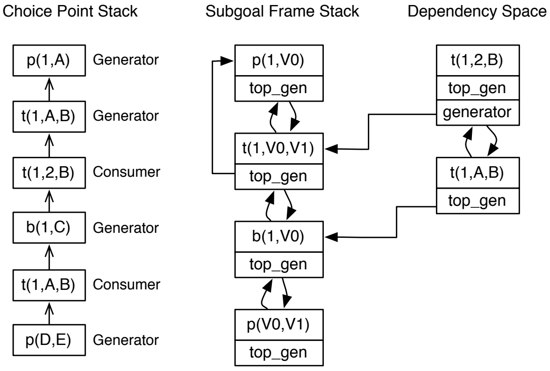


Figure 5.10: Evaluation before pruning the subsumed subgoal p(1,A).

The generator t(1,A,B) is internal to p(1,A) and has two external consumers: one variant and one subsumed consumer, t(1,2,B). Here, we modify the state of subsumptive subgoal t(1,A,B) to *pruned* and turn each external consumer node into a retroactive node. We are hoping that t(1,A,B) will be reactivated and that t(1,2,B) will still consume from its initial generator. Figure 5.11 shows the state of computation after pruning.

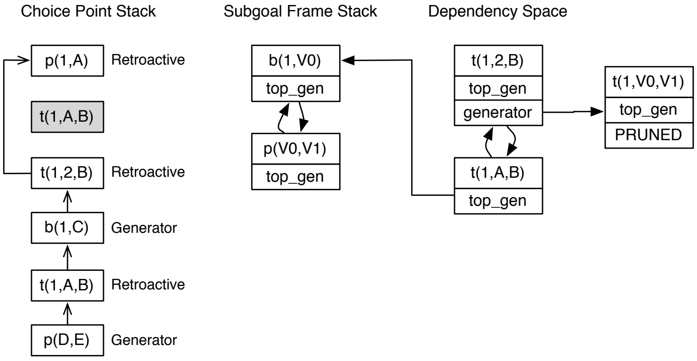


Figure 5.11: Evaluation after pruning the subsumed subgoal p(1,A).

Next, evaluation of p(D,E) generates a call to the subsumptive subgoal t(D,A,B), which is a generator subgoal. By forward execution, we call b(F,G) which subsumes b(1,C) and triggers a new external pruning operation. The computation of b(1,C) contains the internal consumer t(1,A,B) that is currently associated with a retroactive node. We delete this consumer from the dependency space and as t(1,VAR0,VAR1) has no more variant consumers, we update its state to *dead*, in order to inform the retroactive nodes that may dependent on it that it will no longer produce answers (in this example, t(1,2,B)). The result of pruning is shown in Figure 5.12.

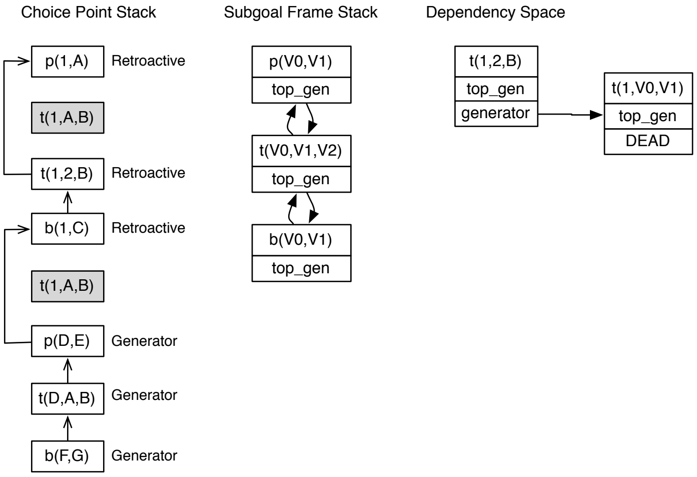


Figure 5.12: Evaluation after pruning the subsumed subgoal b(1,C).

After the subgoals b(F,G), t(D,A,B) and p(D,E) complete, we backtrack to the retroactive node b(1,C). This node is transformed into a loader node as the generator subgoal has already completed. After the answers have been exhausted, we backtrack to the retroactive node t(1,2,B). As the current generator subgoal is *dead* we transform the consumer subgoal frame into a generator subgoal frame and the retroactive node is transformed into a generator node, because the subgoal must now generate its own answers. In order for this to work, the answer template is transformed from {1, 2, B} to {B}. Figure 5.13 shows the state of the computation after this transformation.

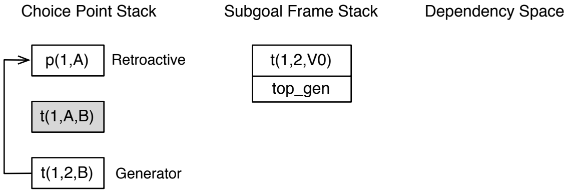


Figure 5.13: Evaluation after transforming the subsumed consumer t(1,2,B) in a subsumptive generator.

5.3.5 Orphaned Consumers

An orphaned consumer is an external consumer that loses its generator after a pruning operation. As we have seen, we transform each orphaned consumer node into a retroactive node. When an orphaned consumer node is reached by means of backtracking it will be transformed into either: (1) a loader node, if the pruned generator was reactivated and has completed; (2) a consumer node, if the pruned generator was reactivated but has not completed yet, which means that below the new reactivated subgoal choice point there is a dependency to an upper generator node, thus this new consumer will participate in the completion operation as usual; or (3) a generator node, if the pruned generator was not reactivated until then. This latter situation only occurs with variant or subsumptive tabling. With retroactive tabling, the execution of a subsuming subgoal that prunes a generator call G, will necessarily call in its evaluation the same or a more general than G, and the subsuming subgoal will update the **generator** field of any subsumed subgoal frame, including orphaned consumers.

By default, orphaned consumers always keep their frames on the dependency frame stack. The frame is only removed if the retroactive node turns into a loader or generator node. If the retroactive node turns again into a consumer node, lazy removal of dependency frames allows us to avoid removing and allocating a new frame and the potentially expensive operation of inserting it on the dependency frame stack in the correct order (ordered by choice point address).

5.3.6 Lost Consumers

While it is usually possible to transform each retroactive node into the correct type of node by means of backtracking or answer resolution, there are some cases where this is not possible. This maybe the case of external consumers turned into retroactive nodes and we call them *lost consumers*.

For an example, consider the program in Figure 5.14 and the query goal '`a(X,Y)`'.

Evaluation starts by storing a generator node for `a(X,Y)` and then by calling the retroactive subgoal `p(1,X)`. Next, the subgoal `b(1,X)` is called, creating a new generator node, and then `b(1,X)` calls subgoal `a(_,Y)`, that is a variant of the initial subgoal and thus a consumer node is stored. As no answers are available to consume, evaluation suspends and then backtracks to the second clause of `b/2`, but it does not unify with `b(1,X)`. By backtracking, we attempt the second clause of `a/2`, which

```
:- use_variant_tabling [a/2, b/2].
:- use_retroactive_tabling p/2.

a(X, 0) :- p(1, X).
a(0, Y) :- b(1, Y).
a(X, Y) :- p(X, Y).

b(1, Y) :- a(_, Y).
b(2, 1).

p(X, Y) :- b(X, Y).
```

Figure 5.14: An example of a program with a lost consumer.

originates a call to `b(1,Y)`. This is a variant call of `b(1,X)` and a new consumer is created. This node must be suspended as no answers are available (gray node in Figure 5.15).

Through backtracking, we execute the third clause of `a/2` and the retroactive subgoal `p(X,Y)` is called. As this subgoal subsumes `p(1,X)` we must prune the evaluation of `p(1,X)`. The internal generator `b(1,X)` is pruned, leaving an orphaned consumer `b(1,Y)`, and the internal consumer `a(_,X)` is simply thrown away. Note that here, we do not have a frontier choice point to prevent the evaluation to step into the pruned branch when backtracking. This is safe, because the branch including the subsumed subgoal will only be resumed on consumers during completion and thus no backtracking to previous choice points will occur.

After `p(X,Y)` is fully explored, the following answers are generated for the subgoal `a(X,Y)`: {`X = 1`} and {`X = 2, Y = 1`}. Then, we attempt completion at the leader node, `a(X,Y)`. At this point, `b(1,Y)` still remains a retroactive node and it is clear that it must be resumed in order to be reactivated as a generator, and consequently, generate more answers to `a(X,Y)`, namely {`X = 0, Y = 1`} and {`X = 0, Y = 1`}. However, since `b(1,Y)` is not a real consumer, it will not participate in the completion operation before retroactive resolution is applied. Therefore, to ensure that all retroactive nodes are resumed, the completion operation is extended to, while traversing the dependency space checking for new answers, also *check for retroactive nodes*, and resume the corresponding node in both cases. In the example, `b(1,Y)` is resumed and transformed into a generator node, thus allowing the computation to finish correctly.

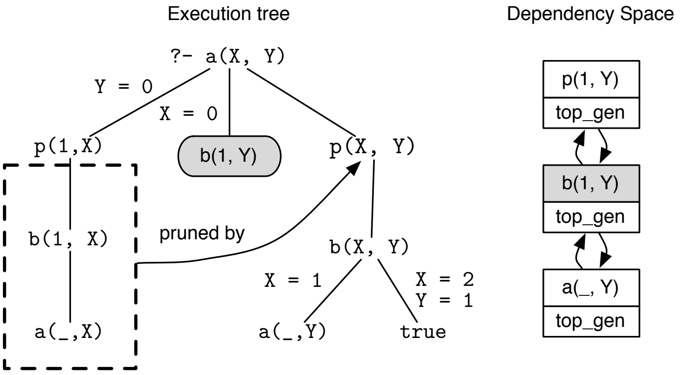


Figure 5.15: Lost consumer `b(1,Y)` after an external pruning.

5.3.7 Pseudo-Completion

When a subsumed subgoal G is pruned being a leader node for some consumers, the completion operation will not be run at node G because G is now a retroactive node thus, it might happen that these consumers will not be resumed by other leader nodes. Therefore, we must ensure that every consumer is resumed in order to fully explore every evaluation branch.

Consider the query goal '`p(1,A,B), p(1,3,C), p(D,E,F)`' and the program in Figure 5.16. The call to `p(1,A,B)` first succeeds with the answer {`A = 2, B = 3`}. Next, subgoal `p(1,3,C)` is called and a first consumer for the subsuming call `p(1,VAR0,VAR1)` is created (consumer $C1$ in Figure 5.17). As no matching answers are available to consume, execution backtracks to `p(1,A,B)` and a second answer is generated, {`A = 3, B = 2`}. Next, subgoal `p(1,3,C)` is called again and a new consumer for `p(1,VAR0,VAR1)` is stored (consumer $C2$ in Figure 5.17). This consumer consumes the answer {`C = 2`} and execution proceeds.

```
:- use_retroactive_tabling p/2.

p(1, 2, 3).
p(1, 3, 2).
```

Figure 5.16: An example of a program executing pseudo-completion.

Subgoal `p(D,E,F)` is then called and the evaluation of `p(1,A,B)` must be pruned (Figure 5.17). The subgoal frame for the subgoal `p(1,A,B)` is transformed into a consumer subgoal frame, and the subgoal frames for the subgoals `p(1,3,C)` and `p(1,A,B)` have the **generator** field made to point to the subgoal frame of `p(D,E,F)`.

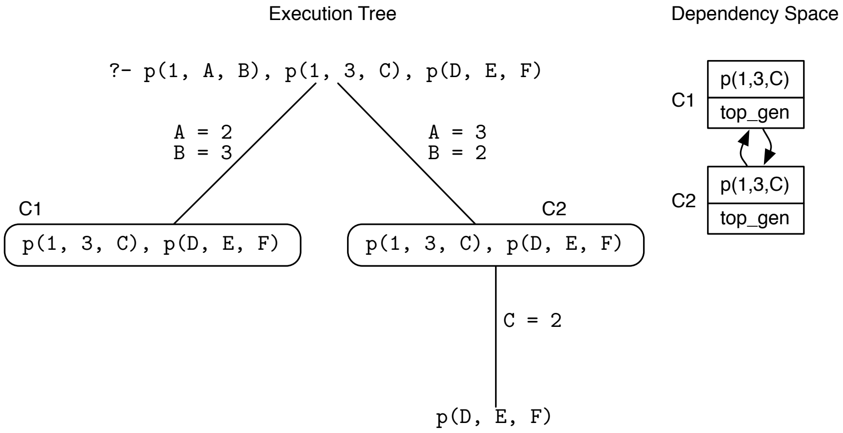


Figure 5.17: Evaluation before pruning the subsumed subgoal `p(1,A,B)`.

After `p(D,E,F)` completes, execution backtracks to retroactive node $C2$ that is transformed in a loader node by retroactive resolution. After all answers are loaded, execution backtracks to the retroactive node of `p(1,A,B)`. If we transform the retroactive node into a loader node and then load all the answers relevant to this node we might lose the consumer $C1$, because there is no other way to reach that node.

Notice that when consumer $C1$ has been created its leader node was `p(1,A,B)`. Hence, before loading all answers in `p(1,A,B)`, we act as a *pseudo-leader* (since no upper dependencies exist) and we look for younger retroactive nodes with unconsumed answers on the dependency frame stack. When we find node $C1$, we set the field **backchain_cp** of the its dependency frame to the choice point of the pseudo-leader, and computation is first resumed at $C1$ (Figure 5.18).

After $C1$ is solved through retroactive resolution, it loads all its answers and, instead of backtracking, jumps to the choice point saved in the **backchain_cp** field, thus allowing the pseudo-leader to resume other retroactive nodes in the same situation. The reason to use the **backchain_cp** field instead of backtracking, is because any choice point between the pseudo-leader and the target consumer as been fully exploited, thus we must jump explicitly between nodes. The process of resuming consumers through a pseudo-leader is called *pseudo-completion*.

5.3.8 Leader Re-Computation

Each dependency frame contains the value of the leader node during the creation of the consumer node. External consumers can reference as a leader either: (1) a pruned

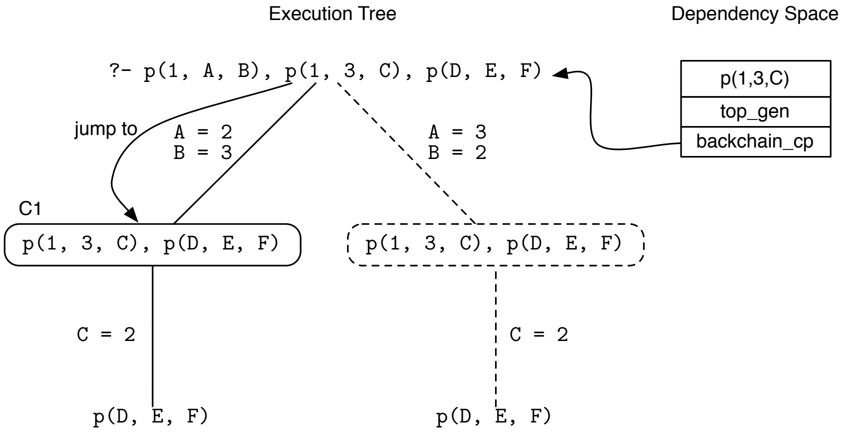


Figure 5.18: Executing a pseudo-completion.

generator choice point; or (2) any other generator. In the second case we keep the dependency frame unmodified. In the first case, we update the leader choice point field (**leader_cp**) to point to the consumer node itself.

The reason we update the **leader_cp** field is to avoid the leader computation algorithm to compute a leader that simply does not exist, making completion impossible. For an example describing this problem, consider the program in Figure 5.19 and the query goal ‘`p(1,A), a(B,C), a(1,D), p(E,F)`’.

```
:- use_retroactive_tabling p/2.
:- use_variant_tabling a/2.

a(1, 3).
a(1, 2).
a(2, 4).

p(X, Y) :- a(X, Y).
```

Figure 5.19: An example of a program leading to the leader re-computation problem.

During the evaluation of the subgoal `p(1,A)`, a generator node for `a(1,A)` is created. Next, a new generator is allocated for `a(B,C)`, followed by a consumer for `a(1,D)`. When `p(E,F)` is called to prune `p(1,A)` (Figure 5.20), the generator `a(1,A)` must also be pruned, but the **leader_cp** field of the consumer `a(1,D)` still points to `a(1,A)`. During execution of `p(E,F)` a new consumer is allocated for `a(E,F)` that will compute that its leader is the pruned `a(1,A)` generator, because below the generator node `a(B,C)` for the new consumer node `a(E,F)` there is a consumer node, `a(1,D)`, with an older leader dependency.

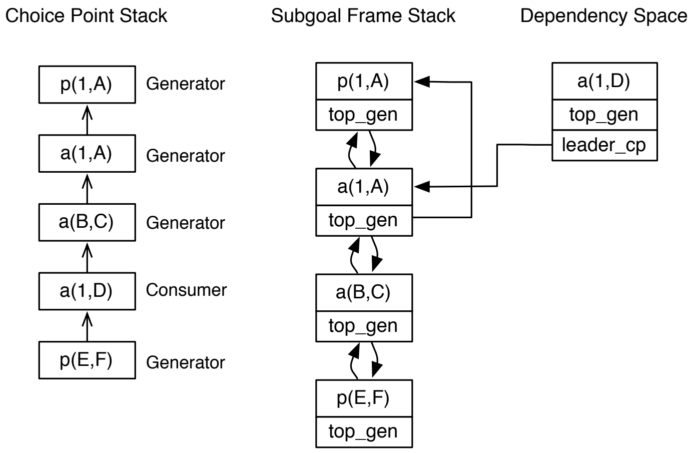


Figure 5.20: Evaluation before pruning the subsumed subgoal `p(1,A)`.

Finally, execution proceeds and completion is attempted at the subgoal `p(E,F)`. Because this is not the leader node, we backtrack to `a(1,D)` that is transformed into a generator and then completes. Next, we backtrack to `a(B,C)` to try other alternatives of `a/2` and then completion is attempted, but without success because the leader node is the pruned generator `a(1,A)`. It is then impossible to complete the evaluation because no leader node will be reached.

By our delineated rules, the `leader_cp` of the consumer `a(1,D)` would have been modified to itself, thus making the generator `a(B,C)` the leader of the computation at that point. Notice that the consumer node, `a(X,Y)` created during the evaluation of `p(E,F)` would still consider that the current leader is `a(B,C)`. Figure 5.21 shows the state of the computation at that point.

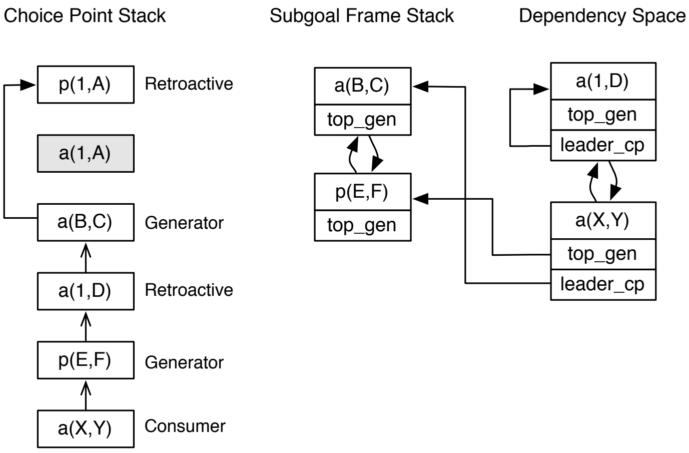


Figure 5.21: Updating the **leader_cp** fields to avoid the leader re-computation problem.

5.4 Internal Pruning

Although all the previous examples use external pruning, both external and internal pruning suffer from the issues described previously. This section explores internal pruning and its differences to external pruning.

Internal pruning occurs when the subsuming subgoal S is internal to the evaluation of the subsumed subgoal R. In this type of pruning we want to keep one part of R running, the one that computes S, hence we are able to compute all answers of R just by computing S.

Our approach involves computing S using local scheduling [FSW96], but without returning answers to the environment of R, as it has been pruned. Instead, we jump directly to the choice point of R, which was transformed into a retroactive node, and resume the computation there in order to consume the matching answers found by S. When resuming the retroactive node for R, it can become either: (1) a loader node, if S has completed; or (2) a consumer node, if S has not completed because it is not a leader node, i.e., the leader node is above R. Notice that, when the completion operation is later attempted at the leader node, the computation can still be resumed, as usual and without any special handling, at R or at the internal consumers of S, until no unconsumed answers are available.

For an example, consider the `path/2` program presented in Figure 5.22. It computes the reachability between two nodes on a directed graph by using a left recursive definition. To know from which nodes we can reach node 3, we are interested in the solutions for the query goal 'path(X,3)'.

```
:- use_retroactive_tabling path/2.

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

edge(1, 2).
edge(2, 3).
```

Figure 5.22: Left recursive path/2 program with retroactive tabling.

Execution starts by creating a generator node for path(X,3), followed by a call to subgoal path(X,Z). Given that path(X,Z) is internal to the computation of path(X,3), we have a case of internal pruning. Using the rules for internal pruning defined above, we will evaluate path(X,Z) with local scheduling (Figure 5.23).

Next, a repeated call to the subgoal path(X,Z) is made and a consumer is created. As no answers are available for consumption, execution backtracks to the second clause of p/2. Here, we call edge(X,Y) and two new answers for path(X,Z) are generated, {X = 1, Z = 2} and {X = 2, Z = 3}. Execution returns to path(X,Z) and completion is attempted. As the path(X,Z) consumer now has answers to consume, they are thus consumed and by forward execution the solution {X = 1, Z = 3} is generated for path(X,Z). Notice that these answers are not returned to the environment of path(X,3), but are only saved on the table space.

After a batch of repeated answers, execution backtracks to path(X,Z) where completion is attempted again. With no more unconsumed answers, the subgoal path(X,Z) completes and instead of backtracking, it jumps directly to the retroactive node of the subgoal path(X,3). Here, the retroactive node first determines the relevant answers from the set of answers generated for path(X,Z), namely, {X = 1} and {X = 2}. Next, the retroactive node is transformed into a loader node, thus loading its answers.

5.4.1 Multiple Internal Pruning

An important aspect in internal pruning is *multiple internal pruning*. Consider that a subgoal R_1 calls recursively internal subgoals $R_1, R_2, ..., R_n$ until a subgoal S is called that subsumes $R_1, R_2, ..., R_n$. In such cases, we ignore all intermediate subgoals and answers are only pushed from S to R_1, the top subgoal. For an example, consider the query goal 'p(1,X)' and the program in Figure 5.24.

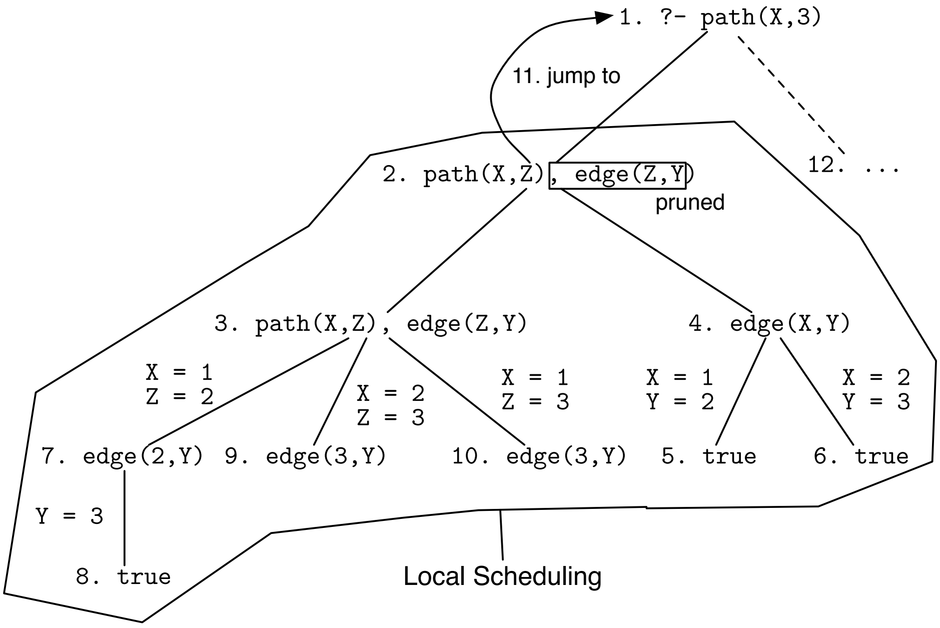


Figure 5.23: Evaluating 'path(X,3)' using retroactive tabling.

```
:- use_retroactive_tabling p/2.

p(1,X) :- p(2,X).
p(2,X) :- p(X, _).
p(2,4).
p(1,5).
```

Figure 5.24: An example of a program illustrating multiple internal pruning.

Execution starts by storing generator nodes for `p(1,X)` and `p(2,X)` and then `p(2,X)` calls `p(X,_)` that subsumes both `p(1,X)` and `p(2,X)`. Pruning is done between the top subsumed subgoal `p(1,X)` and the subsuming subgoal `p(X,_)` and the node for `p(2,X)` is ignored (Figure 5.25). The choice point for `p(1,X)` is transformed into a retroactive node and execution proceeds by applying local scheduling to evaluate `p(X,_)`.

After `p(X,_)` completes with 7 answers, execution is resumed at the retroactive node of subgoal `p(1,X)`, where the answers {`X = 1`}, {`X = 2`}, {`X = 4`}, and {`X = 5`} are loaded (Figure 5.26). Notice that while the subgoal `p(2,X)` did not participate in the later phase of the computation, it has also been completed during the computation of `p(X,_)`.

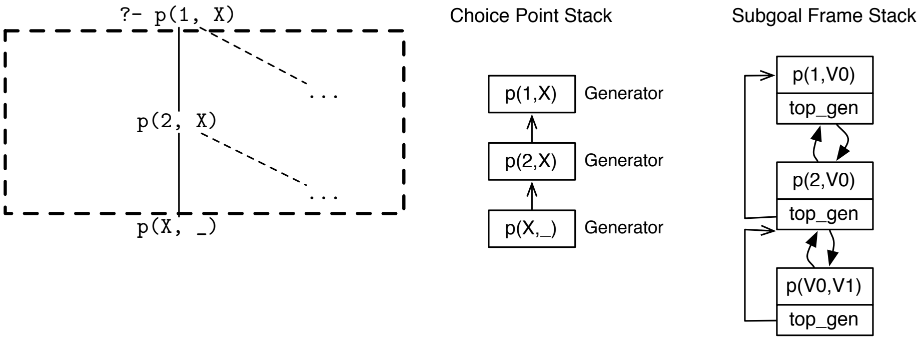


Figure 5.25: Evaluation before multiple internal pruning.

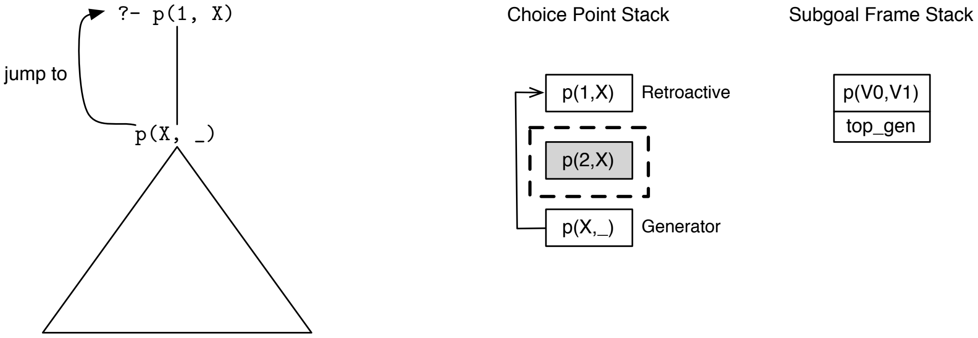


Figure 5.26: After the evaluation of `p(X,_)`.

5.5 Mixing External and Internal Pruning

Although internal and external pruning form the basis of Retroactive Call Subsumption, clear rules must be devised in order to prune multiple subgoals that combine both types of pruning. We want to the minimize the number of pruned subgoals in such a way that a choice point is pruned at most once. Figure 5.27 shows the procedure that implements that idea. This procedure follows the principles of Observation 1 and Observation 2.

Observation 1. *Let $R_1, R_2, ..., R_n$ be a set of subgoals that are recursively internal. In order to prune the subgoals $R_1, R_2, ..., R_n$, only the top subgoal R_1 needs to be pruned.*

Observation 2. *Let G be a subgoal and G' a subgoal internal to G. If G is pruned, G' is also pruned.*

The `prune_subgoal_list` procedure accepts as arguments a subsuming subgoal (as a subgoal frame) and the list of subgoals (as subgoal frames) to prune. This list is obtained by searching for running subsumed subgoals in the subgoal trie (see

```
prune_subgoal_list(subsuming, list) {
  // phase 1: compute oldest internal subgoal to prune
  oldest_internal_sg = NULL
  foreach (subgoal in list)
    if (is_internal(subgoal))
      if (oldest_internal_sg == NULL or
            older_than(generator_cp(subgoal), generator_cp(oldest_internal_sg)))
        oldest_internal_sg = subgoal
      remove_from_list(list, subgoal)
   // change type of subgoal frame and producer
   type(subgoal) = RETROACTIVE_CONSUMER
   generator(subgoal) = subsuming

  // from this point on, only external subgoals in 'list'

  // phase 2: return early by checking the existance of external subgoals
  if (empty(list))
    // no external subgoals
    internal_pruning(subsuming, oldest_internal_sg)
    return

  if (oldest_internal_sg)
    // phase 3: filter external subgoals internal to 'oldest_internal_sg'
    foreach (subgoal in list)
      if (is_internal_subgoal_frame(oldest_internal_sg))
        remove_from_list(list, subgoal)

    // internal prune with 'oldest_internal_sg'
    internal_pruning(subsuming, oldest_internal_sg)

  // phase 4: remove external subgoals that are
  // internal to other subgoals in 'list'
  foreach (subgoal in list)
    if (is_internal_to_set(subgoal, list))
      remove_from_list(subgoal)

  // phase 5: now use the remaining and mutually exclusive
  // external subgoals to apply external pruning
  foreach (subgoal in list)
    external_pruning(subsuming, subgoal)
}
```

Figure 5.27: Pseudo-code for procedure `prune_subgoal_list`.

Section 5.7). Initially, the procedure computes the oldest internal subgoal R in the list of subgoals and discards the other internal subgoals (phase 1). By using the multiple internal pruning principle (Observation 1), we can prune only the oldest subgoal because pruning the oldest subgoal will also prune the other internal subgoals as a side-effect. The `is_internal` function uses an optimization to determine if the subsuming subgoal is internal to the argument subgoal and differs from the `is_internal_subgoal_frame` by not making use of the subgoal dependency tree (see the implementation details section). Notice also that we change the corresponding subgoal frame type of all subsumed subgoals to `RETROACTIVE_CONSUMER` and update the **generator** field to point to the subsuming subgoal.

Next, in phase 2 we check if the list of subgoals is empty, that is, if no external subgoals exist we thus return early from the procedure by doing an internal pruning. In phase 3, we remove all subgoals that are internal to the subgoal R found in phase 1. Since we will prune the computation of R, every subgoal internal to R will also be pruned (except the subsuming subgoal), thus there is no need to prune the two subgoals separately (Observation 2). Next, we do an internal pruning using the R subgoal frame and the resulting list at this point will be either empty or only containing external subgoal frames not internal to R.

In phase 4, we iterate over the resulting list of subgoals and we remove the subgoals that are internal to another one in the list. This phase is also based on Observation 2. Finally, in phase 5 we apply external pruning to each remaining external subgoal.

For an example, consider the program in Figure 5.28 and the query goal '`p(2,X), p(4,Y)`'. Initially, execution creates the generator `p(2,X)` followed by the internal generator `p(3,X)`. Next, the subgoal `p(4,Y)` is called and the first matching alternative calls the subgoal `p(5,Z)`, that immediately generates the answer {`Z = 2`}. In the continuation, subgoal `p(_,Y)` is called, which subsumes all the previous subgoals (Figure 5.29).

```
:- use_retroactive_tabling p/2.

p(2, X) :- p(3, X).
p(2, 1).
p(3, 2).
p(3, 5).
p(4, X) :- p(5, Z), p(_, X).
p(4, 7).
p(5, 2).
p(5, 1).
```

Figure 5.28: An example of a program illustrating multiple pruning.

By following the rules defined above, the oldest internal subgoal is `p(4,X)`. Next, we filter the subgoals that are internal to `p(4,X)`, and the subgoal `p(5,Z)` is removed from the list. Computation is then internally pruned for `p(4,X)`.

The remaining external subgoals are `p(2,X)` and `p(3,X)`. Here we must ignore the subgoals that are internal to each other. The subgoal `p(3,X)` matches these conditions, and is thus removed. Finally, the only remaining subgoal, `p(2,X)`, is then externally

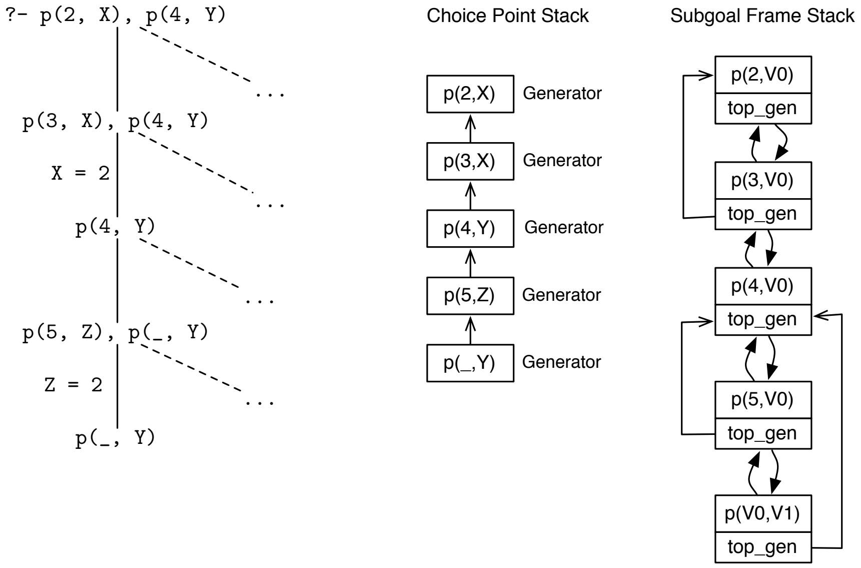


Figure 5.29: Evaluation before multiple internal/external pruning.

pruned and the computation can continue as usual.

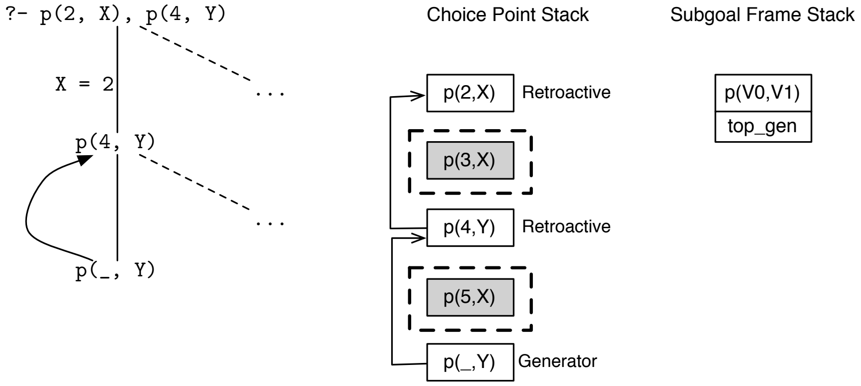


Figure 5.30: Evaluation after multiple internal/external pruning.

5.6 Single Time Stamped Trie

Once a pruned subgoal is reactivated and transformed into a loader or consumer node, it is important to avoid consuming answers that were generated when the subgoal was a generator. In order to efficiently identify what answers have already been used, we designed the *Single Time Stamped Trie* (STST) table space organization. In STST we have a common answer trie to all subgoal calls for the predicate. This approach reduces memory usage because an answer is represented only once and permits easy sharing of answers between subgoals, because an answer can be referenced by various subgoals.

In this new table space organization, each tabled predicate has two tries, the subgoal trie as usual and the STST, while each subgoal frame has an answer return list that references the matching answers from the STST. Figure 5.31 illustrates an example of the new table space organization for a tabled predicate `p/3` with the subgoals `p(1,VAR0)`, `p(2,VAR0)` and `p(VAR0,VAR1)`.

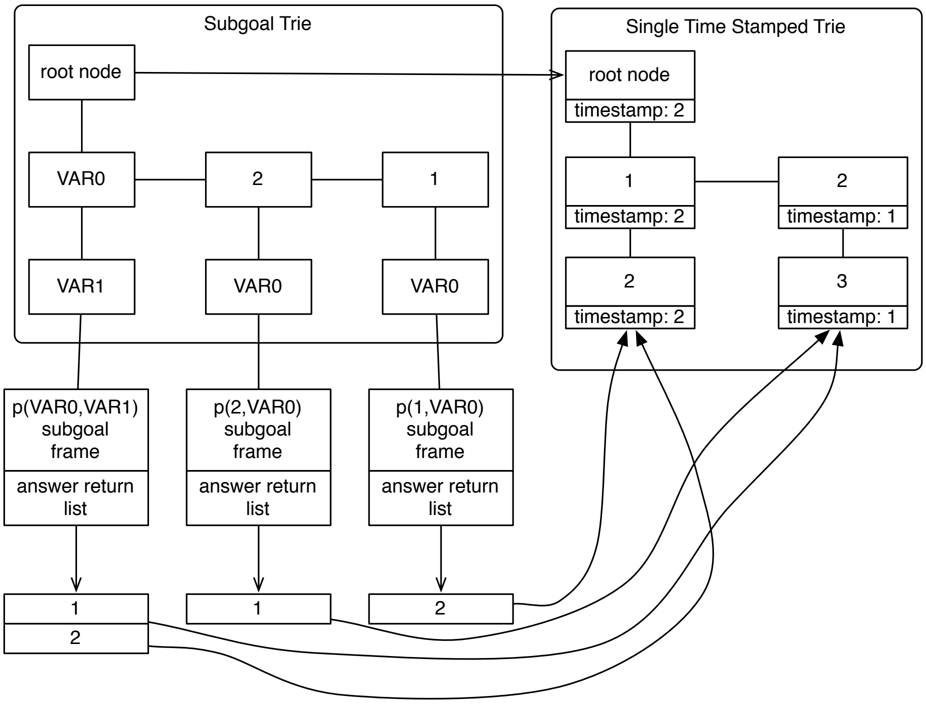


Figure 5.31: STST table organization for the `p/3` predicate.

In terms of implementation, the STST is accessed by using **sibling** field of the root trie node of the subgoal trie. For the subgoal frames, we have extended each retroactive subgoal frame with a **timestamp** field that stores the time stamp of the last answer generated or consumed. At any time, the answers in the answer return list are thus the matching answers from the STST that have a time stamp between 0 and **timestamp**. When we turn a subgoal frame from generator to consumer, we can collect new answers by using the time stamp stored in the subgoal frame, which was the time stamp of the last answer successfully inserted on the STST.

5.6.1 Answer Templates

On a traditional call subsumption engine, that uses an answer trie per generator subgoal, the answer template for each consumer subgoal is built accordingly to its generator subgoal. For *retroactive answer templates*, the answer template is simply built by copying the full set of argument registers for the consumer call. This is a very efficient operation compared to traditional call subsumption.

Notice that we need the full answer template because the answers stored on the STST contain all the predicate arguments, hence the collection and unification of matching answers must be seen as unifying against the most general subgoal.

5.6.2 Reusing Answers

The STST approach also allows reusing answers when a new subgoal is called. As an example, consider that two unrelated (no subsumption involved) subgoals S_1 and S_2 are fully evaluated. If a subgoal S is then called, it is possible that some of the answers on the STST match S even if S neither subsumes S_1 or S_2. Hence, instead of eagerly running the predicate clauses, we start by loading the matching answers already on the STST, which can be enough if, for example, S gets pruned by a cut. This is a similar approach to the *incomplete tabling* techniques for tabling with variant checks [Roc06b].

While the reusing of answers has some advantages, it can also lead to redundant computations. This happens when the evaluation of S generates more general answers than the ones initially stored on the STST. For an example, consider the retroactive tabled predicate `p/1` with only one fact, '`p(X)`'. If `p(1)` is called, the answer represented as {`ARG0 = 1`} is added to the STST and execution would return `true`. If the subgoal `p(X)` is then called, we would search the STST for relevant answers and the first answer would be {`X = 1`}. If we ask for more answers, the system would return a new answer, `true`, and add it to the STST, {`ARG0 = VAR0`} (Figure 5.32). On the other hand, if we called `p(X)` with an empty STST, only the answer `true` would be returned.

5.6.3 Inserting Answers

The insertion of answers on a STST works like the insertion of answers on standard TSTs, but special care must be taken when updating the subgoal frame **timestamp**

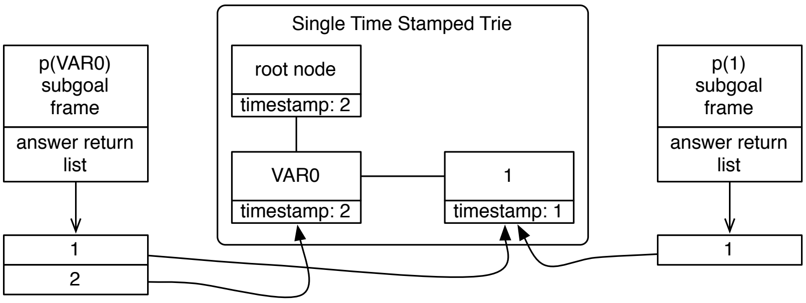


Figure 5.32: Answer redundancy with STST.

field. When only one subgoal is adding answers to the STST, the subgoal frame **timestamp** field is incremented each time an answer is inserted. Repeated answers are easily recognized by testing if the answer is new or not. The problem arises when various subgoals are inserting answers, as it may be difficult to determine when an answer is new or repeated for a certain subgoal.

Let's consider that two subgoals S_1 and S_2 are currently and that S_1 has generated the first 3 answers (time stamp 3) and S_2 has generated answers 4, 5 and 6 (time stamp 6) (see Figure 5.33).

Now, if S_1 generates answer 5, we can incorrectly detect a repeated answer for this subgoal if we consider that repeated answers on the STST are repeated answers for the subgoal (which are not).

An alternative would be to consider it a new answer, since the old S_1 time stamp is in the past ($3 < 5$). But this can also lend to problems after we update S_1 time stamp to either 6 (the global time stamp) or 5 (the newer answer time stamp for the subgoal). Later, if answer 4 is also found for S_1, it will be considered a repeated answer during its insertion. Therefore, we need a more complex mechanism to detect repeated answers per subgoal.

In our new approach, we use a *pending answer index* for each subgoal frame. This index contains all the answers that are older than the subgoal frame time stamp and were still not generated by the subgoal. It is built whenever the STST global time stamp is greater than the current subgoal frame time stamp by collecting all the relevant answers in the STST with a time stamp greater than the current subgoal frame time stamp. Then, whenever an answer with a past time stamp appears, we look up on the pending answer index to check if the answer is there. If so, we consider it a new answer and remove it from the index; if not, we consider it a repeated answer.

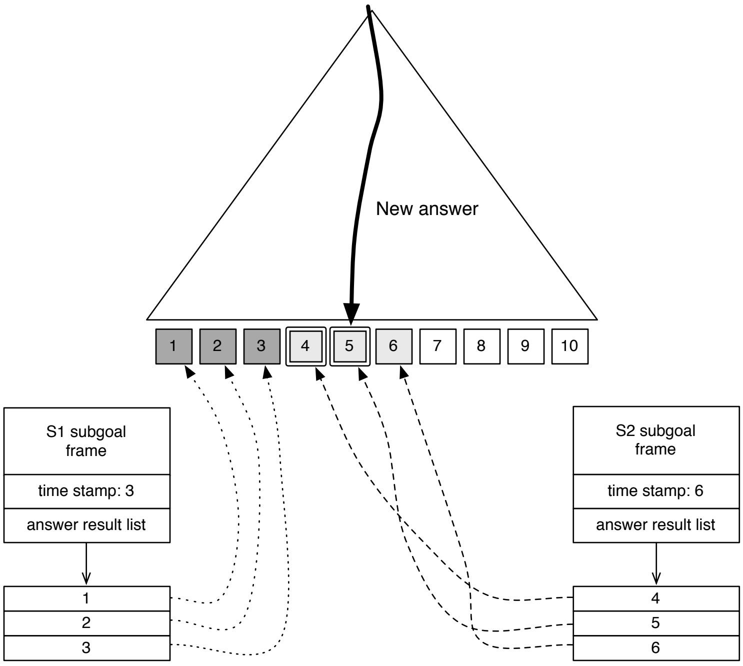


Figure 5.33: STST answer insertion conflicts.

The pending answer index is implemented as a single linked list, but can be transformed into an hash table if the list reaches a certain threshold.

Figure 5.34 presents the code for the `stst_insert_answer` procedure, which inserts an answer on the STST. The pseudo-code is organized into four cases:

1. Answers are inserted in order by the same subgoal. This is the most common situation.
2. The answer being inserted is the only answer that the the current subgoal has still not considered. It is trivially set as a new answer.
3. The time stamp of the answer being inserted is older than the subgoal frame time stamp. The pending answer index must be inspected.
4. The time stamp of the answer being inserted is younger than the subgoal frame time stamp t. We must collect all the relevant answers in the STST with a time stamp greater than t and add them to the pending answer index, except for the current answer.

It is important to note that when a generator subgoal frame is transformed into a consumer subgoal frame, we remove all the answers from the pending answer index

```
stst_insert_answer(subgoal_fr, answer) {
  table_entry = table_entry(subgoal_fr)
  stst = answer_trie(table_entry)
  old_timestamp = timestamp(stst)
  leaf_node = subsumptive_answer_search(stst, answer)
  new_timestamp = timestamp(stst)

  if (new_timestamp == old_timestamp + 1 and
        timestamp(subgoal_fr) == old_timestamp)
    // case 1: new, incremental, answer
    timestamp(subgoal_fr) = new_timestamp
    return leaf_node
  else if (new_timestamp == old_timestamp and
        timestamp(subgoal_fr) == new_timestamp - 1 and
        timestamp(leaf_node) == new_timestamp)
    // case 2: only answer still not considered
    timestamp(subgoal_fr) == new_timestamp
    return leaf_node
  else if (timestamp(leaf_node) <= timestamp(subgoal_frame))
    // case 3: answer with a past time stamp
    // check if it must be considered new
    if (locate_pending_answer(subgoal_sf, leaf_node))
      return leaf_node
    else
      return NULL
  else
    // case 4: answers were inserted by someone else
    pending_list = tst_collect_relevant_answers(stst, timestamp(subgoal_fr),
            answer_template(subgoal_fr))

    remove_from_list(answer_list, leaf_node)
    add_pending_answers(subgoal_sf, pending_list)
    timestamp(subgoal_fr) = new_timestamp
    return leaf_node
}
```

Figure 5.34: Pseudo-code for procedure `stst_insert_answer`.

and we can safely insert them on the answer return list. By doing this, all the consumer mechanisms can be used as usual, without awareness of the pending answer index.

5.6.4 Compiled Tries and Completed Table

Our system only compiles the STST when the most general subgoal is completed. This avoids problems when a subgoal is executing compiled code and another is inserting answers, leading to the loss of answers as hash tables can be dynamically created and expanded.

We also implemented the *completed table optimization.* This optimization throws away the subgoal trie and the subgoal frames when the most general subgoal is completed. When a subgoal call is made, we just build the answer template by copying the argument registers and then we execute the compiled trie, thus bypassing all the mechanisms of locating the subgoal on the subgoal trie, leading to memory and speedup gains.

5.7 Searching Subsumed Subgoals

In order to efficiently find which subsumed subgoals are evaluating and are subsumed by the called subgoal, we extended the subgoal trie data structure and designed a new algorithm to search these subgoals.

5.7.1 Subgoal Trie Data Structure

Each subgoal trie node was extended with a new field, named **in_eval**, which stores the number of subgoals, represented below the node, that are in evaluation. This field is used to, during the search for subsumed subgoals, prune the subgoal trie branches without evaluating subgoals, i.e., the ones with **in_eval** $= 0$.

When a subgoal starts being evaluated, all subgoal trie nodes in its subgoal trie path get the **in_eval** field incremented. When a subgoal completes its evaluation, the path is decremented. Hence, for each subgoal leaf trie node, the **in_eval** field can be equal to either: 1, when the corresponding subgoal is in evaluation; or 0, when the subgoal is completed. For the root subgoal trie node, we know that it will always contain the total number of subgoals being currently evaluated. For an example, consider the subgoal trie in Fig. 5.35 representing four evaluating subgoals and one completed subgoal for a tabled predicate `p/2`.

Consider now, that a new subgoal `p(f(3),5)` enters the evaluation. One new subgoal trie node is created on the subgoal trie to represent the new subgoal call and all the trie nodes from the leaf node to the root node get the **in_eval** field incremented (Fig. 5.36). Note also how the root node is incremented from 4 to 5, meaning that 5 subgoals are now in evaluation.

When a chain of sibling nodes is organized in a linked list, it is easy to select the trie branches with evaluating subgoals by looking for the nodes with **in_eval** > 0. But, when the sibling nodes are organized in an hash table, it can become very slow to

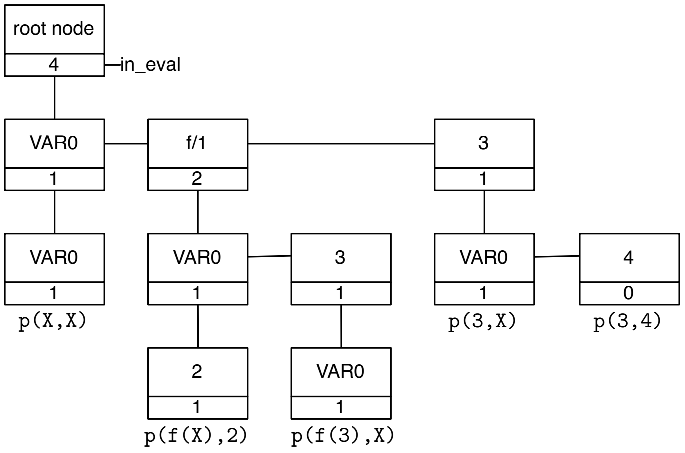


Figure 5.35: The **in_eval** field in a subgoal trie representing a `p/2` tabled predicate

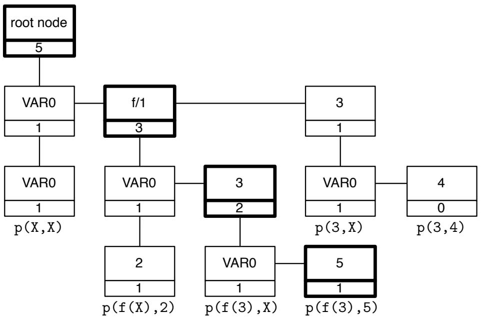


Figure 5.36: Inserting subgoal `p(f(3),5)` in the subgoal trie of Fig. 5.35

inspect each node as the number of siblings increase. In order to solve this problem, we designed a new data structure, called *evaluation index*, in a similar manner to the time stamp index of the TST design.

An evaluation index is a double linked list that is built for each hash table and is used to chain the subgoal trie nodes where the **in_eval** field is greater than 0. Note that this linked list is not ordered by the **in_eval** value. Each evaluation index node contains the following fields: **prev**, a pointer to the previous evaluation index node, if any; **next**, a pointer to the next evaluation index node, if any; **node**, a pointer to the subgoal trie node the index node represents; and **in_eval**, the number of evaluating subgoals under the corresponding subgoal trie node. We also extended the hash table with a field named **index** to point to the evaluation index.

Figure 5.37 shows an hash table and the corresponding evaluation index. Note that an indexed subgoal trie node now uses the **in_eval** field to point to the index node, while a trie node with **in_eval** = 0 is not indexed. To compute the **in_eval** value of a trie node, we first use the **status** field to determine if the node is inside an hash table or not, and then use the **in_eval** field accordingly.

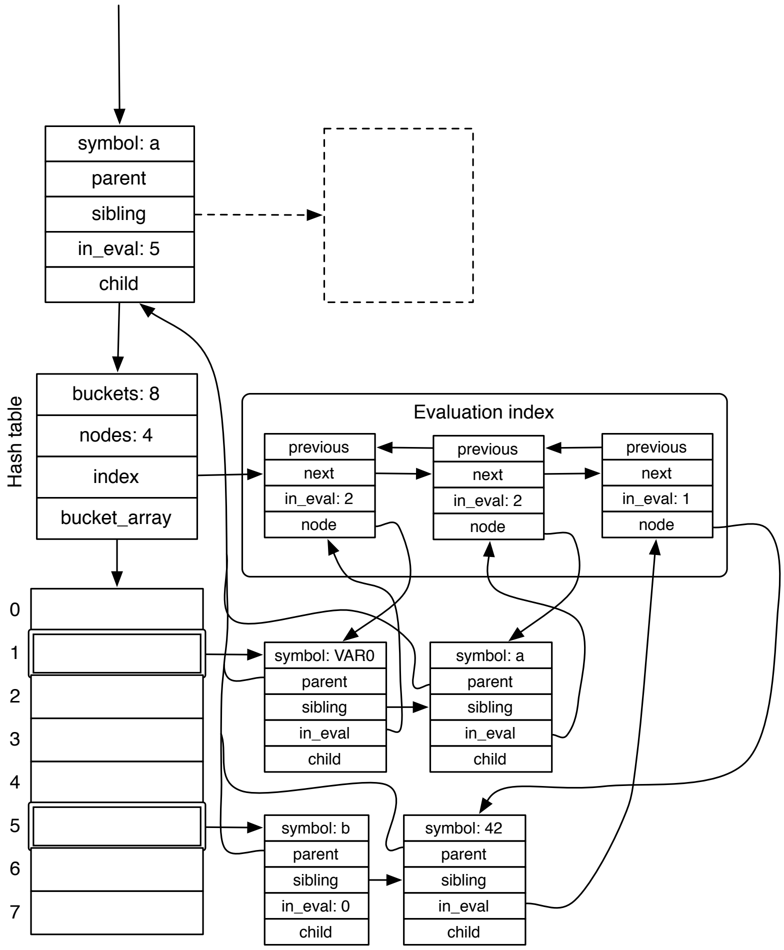


Figure 5.37: An hash table with an evaluation index.

The evaluation index makes the operation of pruning trie branches much more efficient by providing direct access to trie nodes with evaluating subgoals. While advantageous, the operation of incrementing or decrementing a subgoal trie path is more costly, because these indexes must be maintained.

Figure 5.38 presents the pseudo-code for the `increment_in_eval` procedure. This procedure iterates over the subgoal trie path and increments the **in_eval** field from the leaf to the root node. When we find hashed trie nodes, we must check if the node is currently being indexed. If this is the case, we simply increment the **in_eval** field of the index node, otherwise we create a new index node on the evaluation index pointing to the current subgoal trie node and the **in_eval** field of the subgoal trie node is made to point to the index node.

```
increment_in_eval(leaf_node, root_node) {
  current_node = leaf_node
  while (current_node != root_node)
    if (is_hashed_node(current_node))
      if (in_eval(current_node) == 0)
        // not indexed
        hash_table = child(parent(current_node))
        index_node = add_index_node(hash_table, current_node)
        in_eval(current_node) = index_node
      else
        // indexed
        index_node = in_eval(current_node)
        in_eval(index_node) = in_eval(index_node) + 1
    else
      // simple chain list
      in_eval(current_node) = in_eval(current_node) + 1
    current_node = parent(current_node)
  in_eval(root_node) = in_eval(root_node) + 1
}
```

Figure 5.38: Pseudo-code for procedure `increment_in_eval`.

Figure 5.39 shows the resulting hash table and evaluation index if the subgoal trie node in Figure 5.37 that is not indexed gets indexed.

The procedure in Figure 5.40, `decrement_in_eval`, does the inverse job of procedure `increment_in_eval`. When decrementing an indexed subgoal trie node, if the **in_eval** field reaches 0, the trie node no longer needs to be indexed, and hence we must remove the index node from the evaluation index. In the other cases, we simply decrement the respective **in_eval** field.

5.7.2 Matching Algorithm

The algorithm that finds the currently running subgoals that are subsumed by a more general subgoal S works by matching the subgoal arguments SA of S against the trie symbols in the subgoal trie T. By using the **in_eval** field as described previously, we can prune irrelevant branches as we descend the trie. When reaching a leaf node, we append the corresponding subgoal frame in a result list that is returned once the process finishes. If the matching process fails at some point or if a leaf node was reached, the algorithm backtracks to try alternative branches, in order to fully explore the subgoal trie T.

When traversing T, trie variables cannot be matched against ground terms of SA.

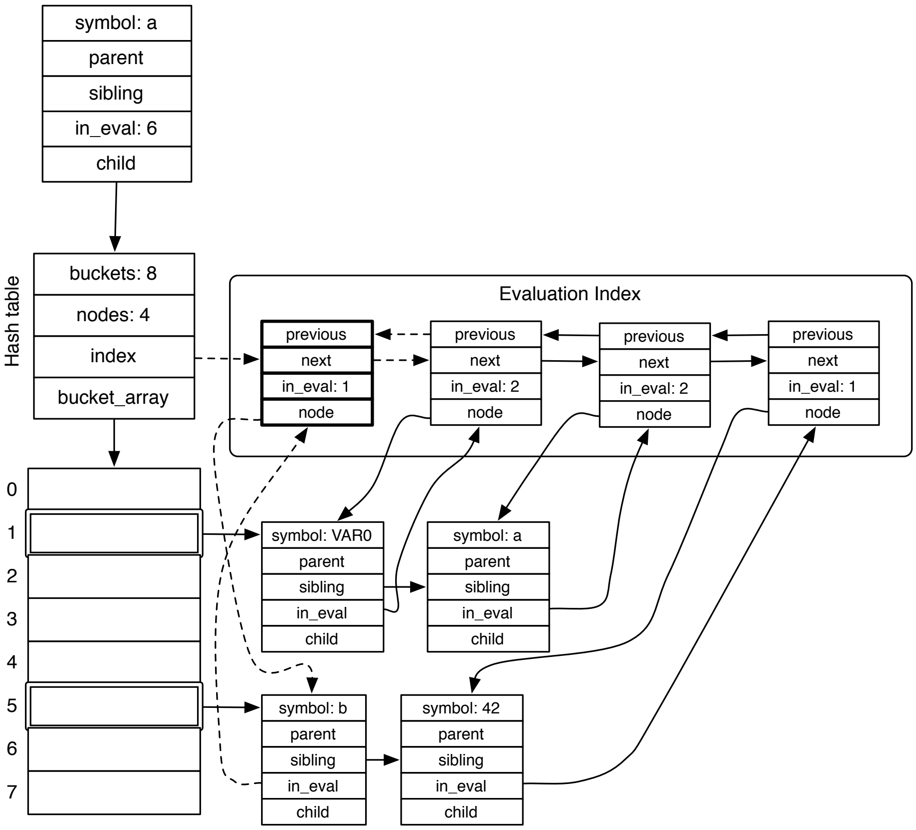


Figure 5.39: Indexing a non-indexed subgoal trie node through an evaluation index.

Ground terms of SA can only be matched with ground terms of T. For example, if matching the trie subgoal `p(VAR0,VAR1)` with the subgoal `p(2,X)`, we cannot match the constant `2` against the trie variable `VAR0`, because `p(2,X)` does not subsume `p(X,Y)`.

When a variable of SA is matched against a ground term of T, subsequent occurrences of the same variable must also match the same term. As an example, consider the trie subgoal `p(2,4)` and the subgoal `p(X,X)`. The variable `X` is first matched against `2`, but the second matching, against `4`, must fail because `X` is already bound to `2`.

Now consider the trie subgoal `p(VAR0,VAR1)` and the subgoal `p(X,X)`. Variable `X` is first matched against `VAR0`, but then we have a second match against a different trie variable, `VAR1`. Again, the process must fail because `p(X,X)` does not subsume `p(X,Y)`. This last example evokes a new rule for variable matching. When a variable of SA is matched against a trie variable, subsequent occurrences of the same variable must always match the same trie variable. This is necessary, because the found subgoals must be *instances* of S. Therefore, this problem can be reduced to the task of finding

```
decrement_in_eval(leaf_node, root_node) {
  current_node = leaf_node
  while (current_node != root_node)
    if (is_hashed_node(current_node))
      index_node = in_eval(current_node)
      if (in_eval(index_node) == 1)
        // remove from index
        hash_table = child(parent(current_node))
        remove_index_node(hash_table, index_node)
        in_eval(current_node) = 0
      else
        // keep indexed
        in_eval(index_node) = in_eval(index_node) - 1
    else
      // simple chain list
      in_eval(current_node) = in_eval(current_node) - 1
    current_node = parent(current_node)
  in_eval(root_node) = in_eval(root_node) - 1
}
```

Figure 5.40: Pseudo-code for procedure `decrement_in_eval`.

all instances of S in trie T. To implement this algorithm, we use the following data structures:

- *WAM data structures*: heap, trail, and associated registers. The heap is used to build structured terms, in which the subgoal arguments are bound. Whenever a term variable is bound, we trail it using the WAM trail;
- *term stack*: stores the remaining terms to be matched against the subgoal trie symbols;
- *term log stack*: stores already matched terms from the term stack and is used to restore the state of the term stack when backtracking;
- *variable enumerator vector*: used to mark the term variables that were matched against trie variables;
- *choice point stack*: stores choice point frames, where each frame contains information needed to restore the computation in order to search for alternative branches.

Figure 5.41 shows the pseudo-code for the procedure that traverses a subgoal trie and collects the set of subsumed subgoals of a given subgoal call. This procedure can be summarized in the following steps:

1. setup WAM machinery and push subgoal arguments into the term stack.
2. fetch a term T from the term stack;
3. search for a trie node N where the **in_eval** field is not 0.
4. search for the next node with a valid **in_eval** field to be pushed on the choice point stack, if any;
5. match T against the trie symbol of N;
6. proceed into the child of N or, if steps 3 or 5 fail, backtrack by popping a frame from the choice point stack and use the alternative trie node;
7. once a leaf is reached, add the corresponding subgoal frame to the resulting subgoal frame list. If there are choice points available, backtrack to try them;
8. if no more choice point frames exist, return the found subsumed subgoals.

5.7.3 Choice Point Stack

To store alternative branches for exploration, we use a choice point stack. Each choice point frame (see Figure 5.42) stores the following fields: **alt_node**, the alternative node to explore; **term_stack_top**, the top of the term stack; **term_log_stack_top**, the top of the term log stack; **trail_top**, the current trail position; and **saved_HB**, the register HB. Note that we used the same choice point stack from Section 4.4.1.

The HB register is used to detect conditional bindings in same manner as for the HB register in WAM choice points, that is, we use it do know if a term variable needs to be trailed. When a choice point frame is popped from the stack, the state of the computation is restored by executing the following actions:

- the current node and parent node are reset;
- all terms stored in the term log stack are pushed back to the term stack;
- the trail is unwound to reset the variables that were bound after choice point creation;
- registers H and HB are reset to their previous values.

```
collect_subsumed_subgoals(subgoal_trie, subgoal_call) {
  save_wam_registers()
  push_arguments(term_stack, subgoal_call)
  subgoals = NULL
  parent = subgoal_trie
  node = child(parent)
  while (true)
    term = deref(pop(term_stack))
    if (is_atom(term) or is_integer(term))
      try_node = try_constant_term(term, node)
    else if (is_functor(term) or is_list(term))
      try_node = try_structured_term(term, node)
    else if (is_variable(term))
      try_node = try_variable_term(term, node)
    if (try_node != NULL)
      push(term_log_stack, term)
      parent = try_node
      node = child(parent)
      if (empty(term_stack))                // new subsumed subgoal found
        add_subgoal(subgoals, subgoal_frame(parent))
      else
        continue
    if (empty(choice_point_stack))
      unwind_wam_trail()
      restore_wam_registers()
      return subgoals
    else
      node = pop_choice_point_frame(choice_point_stack)
      parent = parent(node)
}
```

Figure 5.41: Pseudo-code for procedure `collect_subsumed_subgoals`.

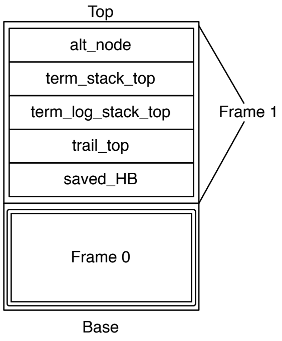


Figure 5.42: Choice point stack organization.

Since constant and structured terms can have at most one matching alternative in a trie level, choice point frames are only pushed when the current term is a variable. Remember that if a node satisfies the **in_eval** requisite, variable terms can match all types of trie symbols, including trie variables.

5.7.4 Matching Constant and Structured Terms

If the next term from the term stack is a constant or a structured term we must match it against a similar ground term only. Both constant and structured terms work pretty much the same way, except that for a list or a functor term we push the term arguments into the term stack before descending into the next trie level. The arguments are pushed into the stack in order to be matched against the next trie symbols.

Figure 5.43 presents the pseudo-code for the `try_structured_term` procedure. This procedure is divided into two steps. We arrive at step 1, if the current node is an hash table. Here, first, we hash the term to get the hash bucket that might contain the matching trie node. If the bucket is empty we simply return `NULL`. Otherwise, we move into step 2. In step 2 we traverse a chain of sibling nodes (a simple chain or a bucket chain) looking for a node with a matching symbol and with a valid `in_eval` value.

```
try_structured_term(term, current_node)
  if (is_hash_table(current_node))
    // step 1: check hash bucket
    hash_table = current_node
    current_node = bucket_array(hash_table, term)
    if (current_node == NULL)
      return NULL
  foreach (node in current_node)
    // step 2: traverse chain of sibling nodes
    if (symbol(node) == term)
      if (in_eval(node) > 0)
        push_arguments(term_stack, term)
        return node
      else
        // no running subgoals below
        return NULL
  return NULL
}
```

Figure 5.43: Pseudo-code for procedure `try_structured_term`.

5.7.5 Matching Variable Terms

A variable term can potentially be matched against any trie symbol. It is only when the variable is matched against a trie variable that the process may fail. Figure 5.44 shows the pseudo-code for the `try_variable_term` procedure. It is defined by three main cases, depending on the type of the current node,namely:

1. the node is an hash table. For faster access of valid trie nodes, we use the evaluation index, which gives us all the valid trie nodes in a linked list. We set the next alternative node to be pushed on the choice point stack by using the function `next_valid_index_node` that uses the **next** pointer of the first index node to locate the alternative trie node.

2. the node is an hashed node, thus is on the evaluation index of the corresponding hash table. In this case, we also use the `next_valid_index_node` function to identify the next alternative trie node.

3. the node is part of a simple linked list. Here we must use the function `next_valid_node` to find the next valid trie node (**in_eval** > 0). The alternative trie node is also set using this function on the sibling node.

```
try_variable_term(variable, current_node) {
  if (is_hash_table(current_node))
    // case 1: hash table
    hash_table = current_node
    index_node = index(hash_table)
    if (index_node == NULL)
      // no running subgoals below
      return NULL
    current_node = node(index_node)
    alt_node = next_valid_index_node(current_node)
  else if (is_hashed_node(current_node))
    // case 2: indexed node
    alt_node = next_valid_index_node(current_node)
  else
    // case 3: simple chain list
    current_node = next_valid_node(current_node)
    if (current_node == NULL)
      return NULL
    alt_node = next_valid_node(current_node)
  push_choice_point_frame(choice_point_stack, alt_node)
  if (try_variable_matching(variable, symbol(current_node)))
    return current_node
  else
    return NULL
}
```

Figure 5.44: Pseudo-code for procedure `try_variable_term`.

After the current valid node and alternative node are set, we push the alternative into the choice point stack and call the `try_variable_matching` procedure (Figure 5.45) to match the term variable with the trie node symbol.

Matching a term variable with a trie symbol depends on the type of the trie symbol. If the trie symbol is a trie variable, we have two cases. If the term variable is free

```
try_variable_matching(symbol, variable, node) {
 if (is_variable(symbol))
   if (is_in_variable_enumerator_vector(variable))
     enumerator_index = enumerator_index(variable)
     var_index = var_index(symbol)
     return enumerator_index == var_index
   else
     // new term variable
     var_index = var_index(symbol)
     mark_variable_enumerator_vector(variable, var_index)
     return TRUE
 else
   // non-variable symbol
   if (is_in_variable_enumerator_vector(variable))
     // variable must be matched against the same trie variable
     return FALSE
   if (is_constant(symbol))
     bind_and_conditionally_trail(variable, symbol)
   else if (is_functor(symbol) or is_list(symbol))
     term = create_heap_structure(symbol)
     bind_and_conditionally_trail(variable, term)
     push_arguments(term_stack, deref(variable))
   else
     return FALSE
}
```

Figure 5.45: Pseudo-code for procedure try_variable_matching.

(i.e., this is its first occurrence), we simply make it to point to the position on the variable enumerator vector that corresponds to the trie variable index and we trail the term variable using the WAM trail. Otherwise, the term variable is already matched against a trie variable (on the variable enumerator vector), thus we get both indexes (term and trie variable indexes) and the matching succeeds if they correspond to the same index (same variable).

If the trie symbol is a ground term, we must verify if the term variable is on the variable enumerator vector and, in such case, we must fail. Term variables matched against trie variables must only be matched against the same trie variable. Otherwise, for constant trie symbols, we simply bind the term variable to the trie symbol. For structured terms (lists and functors), we create the structured term on the heap, bind the term variable to the heap address, and push the new term arguments into the term stack to be matched against the next trie symbols.

5.7.6 Example Execution

Consider the subgoal trie with three executing subgoals represented in Figure 5.46. We want to retrieve subgoals that are subsumed by the subgoal `p(X,2,X)`. Initially, the algorithm setups the WAM registers and then pushes the subgoal arguments into the term stack resulting in the following stack configuration (from bottom to top): `[X,2,X]`.

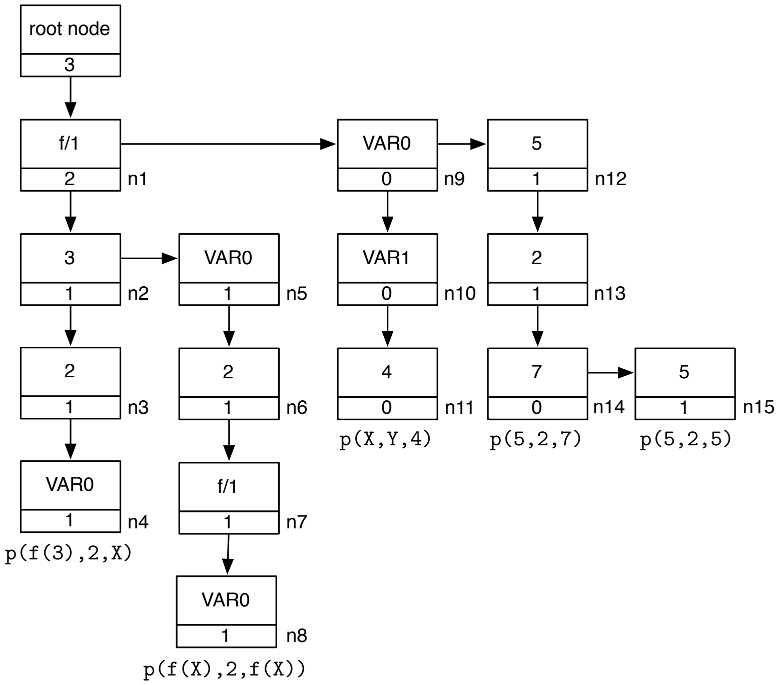


Figure 5.46: An example subgoal trie with three evaluating subgoals.

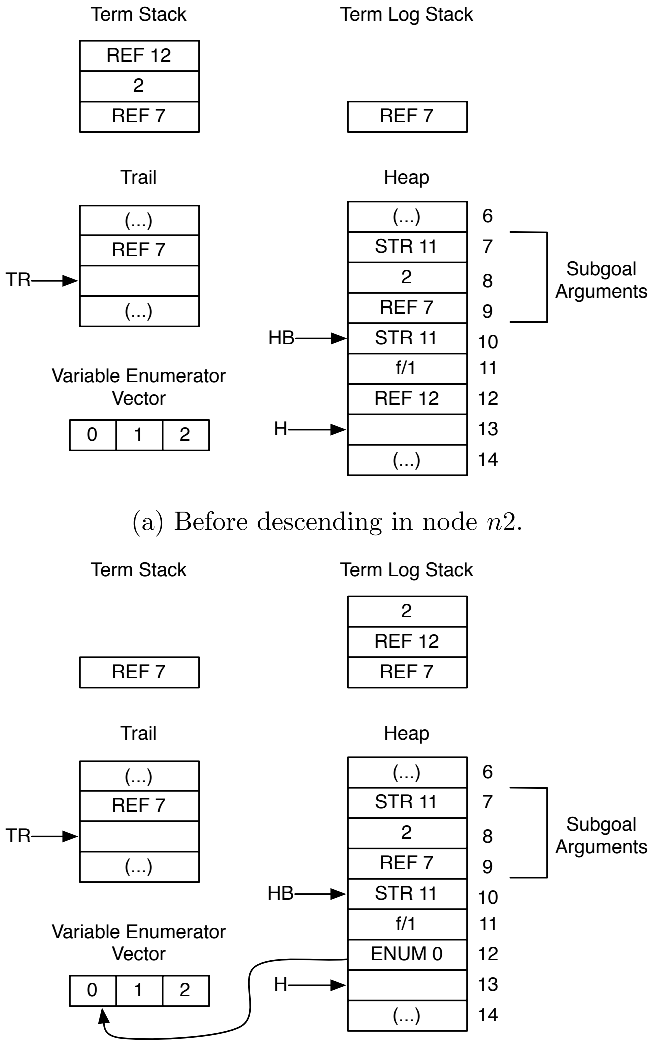
Next, we pop the `X` variable from the term stack and inspect the linked list of nodes $n1$, $n9$ and $n12$. Because `X` is a variable term we can potentially match this term with any node with **in_eval** > 0. We thus match `X` against the symbol `f/1` from node $n1$ by constructing a new `f/1` functor into the heap and pushing a new variable (the functor argument) into the term stack. Figure 5.47a shows the configuration of the data structures at this point. Notice that the term log stack now contains the `X` variable and the register H now points to the next free heap cell. Before descending into node $n2$, we need to push the alternative node $n12$ into the choice point stack. Note that node $n9$ cannot be used as an alternative because no evaluating subgoals exist in that trie branch.

Next, we pop the unbound functor argument from the term stack and we match it against the symbol 3 from node $n2$. Node $n5$ is pushed into the choice point stack, and we now have the following stack configuration: [$n12, n5$]. We then descend into node $n3$, where the next term from the term stack, 2, matches the trie symbol 2. Here, there are no alternative nodes to explore, and matching proceeds to node $n4$.

In node $n4$ we pop the X variable from the term stack, that when dereferenced, points to the constructed f/1 functor on the heap. As we cannot match ground terms, such as f/1, with trie variables, the process fails. We then pop the the top frame of the choice point stack and search is resumed at node $n5$. The choice point stack now has the following configuration: [$n12$]. Because we backtracked to try node $n5$, the term stack is restored with the functor argument, the constant 2 and the X variable. In node $n5$, the unbound functor argument is popped from the term stack and is made to point to the index 0 of the variable enumerator vector, because the trie symbol is the VAR0 trie variable. We then descend into node $n6$, where matching succeeds and then we arrive at node $n7$. Figure 5.47b presents the configuration of the auxiliary data structures at this point. Notice that the variable at the 12^{th} cell now points to a variable enumerator position and that the HB register points to the 11^{th} cell on the heap, which corresponds to the value of the H register when we pushed the top choice point frame on the choice point stack (node $n12$).

Node $n7$ contains the functor f/1 and the next term from the term stack is the X variable that is bound to the functor f/1 created on the heap. Matching therefore succeeds and we get into node $n8$. In node $n8$ we have the trie variable VAR0 and a variable that is in the variable enumerator vector. Because they both correspond to the index 0, matching succeeds and a subsumed subgoal is found: p(f(X),2,f(X)).

Next, we pop the top choice point frame from the choice point stack and search is thus resumed at node $n12$. The term stack is restored to its initial state and the choice point stack is now empty. Node $n12$ contains the trie symbol 5 that matches the term variable X by bounding X to 5. Execution proceeds to node $n13$ where the trie symbol 2 matches the constant term 2. We descend again, now to node $n14$. Here, only node $n15$ can be used. We then pop the variable X from the term stack that is a bound to the constant 5 and matches the trie symbol 5. A new subsumed subgoal is thus found: p(5,2,5). Finally, as there are no more alternatives, the algorithm ends and returns the two found subgoals.



(a) Before descending in node $n2$.

(b) After descending in node $n7$.

Figure 5.47: Auxiliary data structures configuration.

5.8 Implementation Details

In this section we describe some details of the implementation that were not described in the previous sections, but are still important to understand how RCS was implemented.

5.8.1 Faster External or Internal Test

While the subgoal dependency space works is a reasonably fast internal/external test for subgoals, it is possible to implement a faster test that are currently running, e.g, to test if the more general subgoal is internal or external to a target subsumed subgoal.

This mechanism extends each subgoal frame with two new fields: **start_code** and **end_code**, that initially are standard WAM variables. The **start_code** field is bound to an arbitrary value when the subgoal starts executing and unbound when the subgoal backtracks, therefore allowing us to detect if the subgoal is in the current branch. The **end_code** field is bound when a new answer is found, i.e., when the code has reached the end of a program clause, thus allowing us to easily detect if a pruned subgoal is external or external.

5.8.2 Transforming Consumers Into Generators

In order to be able to transform a consumer node into a retroactive node and then into a generator node, all consumer choice points are allocated, by default, as generator choice points.

A generator choice points extends a WAM choice point with two fields: **sg_fr**, a pointer to the subgoal frame; **dep_fr**, a pointer to a dependency frame, if evaluating using local scheduling. Allocated along the generator choice point we also have the answer template and the subgoal arguments. In order for the consumer choice to be the same size as the generator choice point, we extended the consumer choice point with the **sg_fr** field and the subgoal arguments are also pushed into the local stack.

5.8.3 Reference Counting

We extended the subgoal frame data structure with a field called **num_deps**. This field counts the number of dependency frames dependent on this subgoal. Therefore, while updating external consumers to transform them into retroactive nodes we count the number of external consumers already processed, and thus we can exit early when the count reaches the target number. This speedups the pruning process because we known when all external consumers were already processed.

For subsumptive generator subgoal frames, we extended them with an extra field called **num_sub_deps**, that counts the number of subsumed consumers. By knowing how many variant consumers and subsumed consumers we have for a generator we can decide more efficiently when to transform external consumers and if we need to transform subsumed consumer nodes to use a generator subgoal frame, if no variant external consumers exist anymore.

5.8.4 Data Structure Modifications

To be able to set a predicate to use retroactive call subsumption, we added another flag to the **mode_flag** bit field of the table entry. In addition to the flags **variant** and **subsumptive**, **retroactive** is also available.

For the dependency frame, we extended it with the bit field **flags**. Among other things, one bit marks if the corresponding consumer is a potential lost consumer, therefore if the completion operation will attempt to resume the consumer, it will check first for that bit flag to execute retroactive resolution in order to transform the node into a generator.

The retroactive subgoal frame was extended with a **try_answer** field. This field is used to consume the available answers from the answer trie that are relevant to the subgoal before attempting to execute the predicate clauses.

We extended both the subgoal and the dependency frame with two new fields, **previous** and **next**, that are used to double link the frames on the respective stacks. These pointers make the operation of removing a frame from the respective stack more efficient.

5.8.5 Tabling Operations

The most important modifications to tabling operations were done in the tabled subgoal call and completion operations. For the tabled subgoal call operation (Figure 5.48) we distinguish the new retroactive evaluation method by calling the procedure `retroactive_subgoal_search`. When we have a new generator call, after storing the generator node, we collect the running subsumed subgoals, we mark ourselves as a running subgoal, and then we prune the found subgoals by calling `prune_subgoal_list`. Next, we attempt to collect new relevant answers from the STST with `collect_available_retroactive_answers` and if the subgoal frame contains any answer we load them by using the `table_try_answer` instruction.

The tabled subgoal call operation must also check if the subgoal frame is in the pruned state. Here, we simply load all the available answers and then call the `table_try_answer` instruction, which will load all the available answers and then execute the program clauses.

When trying to complete by traversing the dependency space for unconsumed answers, we must also check for lost consumers. These consumers must be restarted as generators in order for computation be completed. Figure 5.49 presents the new completion operation. Note that we detect a lost consumer we call the `retroactive_resolution` instruction in order to restart the consumer.

```
tabled_subgoal_call(table_entry, arguments) {
  subgoal_trie = subgoal_trie(table_entry)

  if(method(table_entry) == VARIANT)
    subgoal_frame = variant_subgoal_search(subgoal_trie, arguments)
  else if (method(table_entry) == SUBSUMPTIVE)
    subgoal_frame = subsumptive_subgoal_search(subgoal_trie, arguments)
  else if (method(table_entry) == RETROACTIVE) // NEW
    subgoal_frame = retroactive_subgoal_search(subgoal_trie, arguments)

  if (is_new_generator_call(subgoal_frame))
    store_generator_node(table_entry, arguments, subgoal_frame)
    if (type(subgoal_frame) == RETROACTIVE_GENERATOR)
      subgoals = collect_subsumed_subgoals(arguments, subgoal_trie)
      increment_in_eval(subgoal_frame)
      prune_subgoal_list(subgoal_frame, subgoals)
      collect_available_retroactive_answers(subgoal_frame)
      if (get_first_answer(subgoal_frame))
        // consume pre stored answers
        first = get_first_answer(subgoal_frame)
        load_answer_from_trie(first)
        try_answer(subgoal_frame) = first
        CP_AP(B) = table_try_answer
        goto continuation_instruction()
  else if (state(subgoal_frame) == pruned) // NEW
    // consume pre stored answers
    state(subgoal_frame) = evaluating
    first = get_first_answer(subgoal_frame)
    load_answer_from_trie(first)
    try_answer(subgoal_frame) = first
    CP_AP(B) = table_try_answer
    goto continuation_instruction()
  else if (is_new_consumer_call(subgoal_frame))
    // (...)
  else
    // (...)
}
```

Figure 5.48: Pseudo-code for the new tabled subgoal call operation.

```
completion(generator) {
  if (is_leader_node(generator))
    df = TOP_DEP_FR
    while (younger_than(cons_cp(df), generator))
      // NEW
      if (is_lost_consumer(df))
        consumer = cons_cp(df)
        CP_B(consumer) = generator
        restore_bindings(CP_TR(generator), CP_TR(consumer))
        goto retroactive_resolution(consumer)
      cont = get_next_answer_continuation(dep_fr)
      if (cont)
        // unconsumed answers
        back_cp(df) = generator
        consumer = cons_cp(df)
        restore_bindings(CP_TR(generator), CP_TR(consumer))
        goto answer_resolution(consumer)
      df = next(df)
    perform_completion()
    adjust_freeze_registers()
  backtrack_to(CP_B(generator))
}
```

Figure 5.49: Pseudo-code for the new completion operation.

5.9 Chapter Summary

In this chapter we demonstrated the shortcomings of tabling by call subsumption and presented a new extension that overcomes these limitations. The new extension is called Retroactive Call Subsumption (RCS) and enables full sharing of answers, independently of the order the subgoals are called.

We presented the concept of pruning the execution of a subsumed subgoal by transforming its state from generator to consumer and then pruning its current execution. We also discussed two main types of pruning: internal pruning, for internal subgoals; and external pruning, for external subgoals. We gave examples demonstrating the issues that arise when pruning a range of choice points, followed by our proposed solutions. We also presented new mechanisms to support RCS: a novel table space called Single Time Stamped Trie (STST), where answers are represented only once; and a novel algorithm to find executing subsumed subgoals in a subgoal trie.

In the next chapter, we will analyze the performance of the two tabling engines we implemented for this thesis. First, we will experiment with YapTab_TST, the engine that implements traditional call subsumption, and then with the engine that supports RCS. We will also do some analysis on the table space organization of each engine.

Chapter 6

Experimental Results

This chapter presents a detailed performance analysis of our two subsumption-based tabling engines that we have developed. We divided this chapter into four sections. The first section describes the set of tabled benchmark programs used. The second section evaluates the engine supporting traditional call subsumption that was implemented by integrating the Time-Stamped Tries algorithms and data structures from XSB into Yap. This includes analyzing the memory gains of call subsumption by measuring the size of the answer tries and comparing it to variant-based tabling. In the third section, we evaluate the retroactive-based tabling engine for programs that do not benefit from the new mechanisms and for programs that can take advantage of this new evaluation method. Finally, in the fourth section we evaluate the STST table space overhead in a potentially not so good scenario, where the operations of loading and storing answers are more expensive than usual.

6.1 Benchmark Programs

In order to assess the performance of our tabling engines we used various programs and data sets. We next briefly describe the programs used (see Appendix A for more details).

path: This program computes the reachability between two nodes in a graph. Connections between two nodes are represented by `edge/2` facts. We used the following graph configurations in our tests: **tree**, a binary tree; **chain**, a chain of nodes; **cycle**, a chain of nodes, where the last node connects with the first one;

pyramid, a pyramid like configuration; and **grid**, where nodes are connected in a grid-like fashion. For the `path/2` predicate itself, we used 6 different versions: `left_first`, `left_last`, `right_first`, `right_last`, `double_first`, and `double_last`.

samegen: The `samegen/2` predicate solves the same generation problem. For this program, we used the configurations described above for `path`.

genome: This program computes the set of nodes that are reachable by nodes 1 and 2 in a graph. This is an interesting problem, since it creates lots of subsumed consumers when using call subsumption. We also used the same configurations described above for `path`. To compute reachability this program uses a `left_last` version.

reach: The `reach/2` predicate computes the reachability in a relation graph for a set of model specifications. The benchmark is actually a set of programs originally taken from the XMC project [RRS$^+$00], which is a model checker implemented atop the XSB system. We used two variants of the `reach/2` predicate, `reach_first` and `reach_last`. The following relation graphs where used:

sieve: *sieve* specification defined for 5 processes and 4 overflow prime numbers.

leader: *leader election* specification defined for 5 processes.

iproto: *i-protocol* specification defined for a correct version with a huge window size.

flora: This program was generated by an object-oriented knowledge base language and application development environment known as FLORA-2 [YK00] [1].

fib: This program uses a `fib/2` predicate to compute the Fibonacci number of a given parameter which allows to benchmark the pruning of one subsumed subgoal.

big: This program also uses the `fib/2` predicate, but instead of one, multiple subsumed subgoals are called and pruned. As a parameter, we can input the number of subsumed subgoals to call and prune.

Again, Note that the relevant parts of the code for these programs are presented in Appendix A.

[1] http://flora.sourceforge.net

The environment for our experiments was an Intel Core(TM) 2 Quad 2.66 GHz with 4 GBytes of memory and running the Linux kernel 2.6.31 with YapTab 6.0.3 and XSB Prolog 3.2. The scheduling strategy used by default was batched scheduling.

6.2 Traditional Call Subsumption with TST

In this section, we first evaluate the performance of YapTab against the SLG-WAM, by comparing the gains obtained by using call subsumption instead of variant checks. In the second part of this section, we measure the impact in terms of space of using call subsumption. For this, we compared the number of answer trie nodes created by using variant checks and by using subsumptive checks.

6.2.1 Performance Evaluation

In order to compare the YapTab tabling engine with the SLG-WAM we used the following programs:

- The `path/2` program with all the combinations of versions and data sets and the query goal ‘`?- path(X,Y).`’.
- The `samegen/2` program with all data sets and the query goal ‘`?- samegen(X,Y).`’.
- The `genome/1` program with all the data sets and the query goal ‘`?- genome(X).`’.
- The two versions of the `reach/2` program with the following queries for each relation graphs:
  - sieve: ‘`?- reach(sieve_0(5,4,27,end),Y).`’.
  - leader: ‘`?- reach(systemLeader_0(5,end),Y).`’.
  - iproto: ‘`?- reach(iproto_0(_,_,end),Y).`’.

For each benchmark, we used variant-based tabling and then subsumption-based tabling. Next, we computed the execution time and compared the speedups obtained ($T_{variant}/T_{subsumptive}$) for each engine. The times presented next are the average of 2 runs. Given that YapTab's implementation is largely based on XSB's code to implement the subsumption mechanisms, we expect the speedups to be very similar.

Some potential differences between them will arise because of certain characteristics, namely: the way they implement the tabling algorithms, the WAM engine itself, the compiled trie code, and the handling of answer templates.

Table 6.1 summarizes the average speedups obtained for each program, while Tables 6.2 and 6.3 show the full details, with times and speedups for YapTab and SLG-WAM.

Program	**SLG-WAM**	**YapTab**
	Average Speedup	***Average Speedup***
left_first	0.76	**0.95**
left_last	0.76	**1.00**
right_first	**1.03**	0.92
right_last	**1.07**	0.98
double_first	1.25	**1.46**
double_last	1.34	**1.35**
samegen	**292.66**	0.93
genome	**596.17**	595.55
reach_first	**0.95**	0.89
reach_last	**0.98**	0.95

Table 6.1: Average speedups for call subsumption in SLG-WAM and YapTab.

The first thing we note is that, YapTab has a better speedup than SLG-WAM in 4 benchmarks, while in 6 of them SLG-WAM wins. While SLG-WAM wins here, the speedups for the two engines are very similar, which proves that our integration efforts were largely successful. However, for the `samegen` benchmark the speedups are not very similar at all, the SLG-WAM engine has an average speedup of 292.66 and YapTab only 0.93. This happens because the performance of the variant-based version of SLG-WAM performs very poorly against YapTab, which explains such big differences (see Table 6.3 for details).

The programs `left_first` and `left_last` do not generate any subsumed consumer, therefore they are good benchmarks to assess the overhead of using the subsumption mechanisms. For YapTab, the overhead is minimal with an average speedup of 0.95 for the `left_first` program. Surprisingly, for the `left_last` program the speedup is equal to 1, that is, both variant-based and subsumptive-based engines have the same execution time. The SLG-WAM behavior for these two programs is clearly worst, with an average speedup of 0.76 for both programs.

The programs `right_first` and `right_last` do generate subsumed consumers, as many as the number of `edge/2` facts. Notably, only the SLG-WAM achieves an average speedup bigger than 1 for both programs. These programs show poor speedups because simple facts are faster to evaluate than to use the time stamped trie to collect relevant answers. In particular, the binary tree graph configuration with the `right_first` and `right_last` programs in YapTab has a very poor speedup of 0.77 and 0.31, respectively, which is clearly influencing the average speedups. (see Table 6.2). In the `right_first` benchmark, the time stamped index is created right at the beginning of the program, when the time stamped trie is still empty, and maintained thereafter, but, in the case of the `right_last` program, the indices are only created when the recursive clause is executed, and at that time the trie already contains a considerable amount of answers. We thus modified the subsumptive engine to create the time stamp index from the beginning, and the `right_last` program had considerable better results. Therefore, we argue that the lazy creation of the time stamped index can affect considerably the execution time, for programs where consumers appear when the answer trie already contains lots of answers. The operation of creating the time stamped index and sorting each trie node at each trie level can be very costly when a large number of nodes exist. By experimentation we found that it seems more efficient to maintain the index as the answer trie is being expanded.

For the `double_first` and `double_last` we have attained speedups between 1.25 and 1.46 for both YapTab and SLG-WAM. These benchmarks are more computationally expensive given that they create more dependencies. However, in subsumption-based tabling, the number of dependencies is reduced and thus no code is executed.

The `genome` program shows the best speedup results, with an impressive average speedup of 595.55 for YapTab. In this program, the subgoal `path(2,X)` and `path(1,X)` are called very early in the evaluation which makes all further subgoals calls to `path/2` that are subsumed by these goals as consumers.

For the model checking programs, the results were not so good with very close speedups for YapTab and SLG-WAM.

6.2.2 Memory usage

In this section we measure the size of the table space, for the variant and the subsumptive engine. As a metric, we use the number of allocated answer trie nodes across all answer tries. For this, we used the following benchmarks:

Data		**SLG-WAM**			**YapTab**		
		Var	*Sub*	*Speedup*	*Var*	*Sub*	*Speedup*
left_first	chain	2844	3908	0.73	2588	2416	**1.07**
	cycle	5972	9364	0.64	6768	6056	**1.12**
	grid	13504	15688	0.86	10140	11408	**0.89**
	pyramid	9676	13552	0.71	12764	15012	**0.85**
	tree	136	160	**0.85**	120	144	0.83
left_last	chain	2724	3996	0.68	2620	2476	**1.06**
	cycle	6188	9744	0.64	6880	6104	**1.13**
	grid	13404	15688	0.85	10164	11480	**0.89**
	pyramid	2388	2920	0.82	2896	2692	**1.08**
	tree	128	156	0.82	116	136	**0.85**
right_first	chain	2840	2836	**1.00**	3344	3412	0.98
	cycle	6316	7108	0.89	6652	6888	**0.97**
	grid	18061	15300	**1.18**	15756	18165	0.87
	pyramid	12092	9804	**1.23**	11832	11452	1.03
	tree	160	184	**0.87**	172	224	0.77
right_last	chain	2812	2872	0.98	3692	3644	**1.01**
	cycle	6368	6996	0.91	7528	6580	**1.14**
	grid	19777	13400	**1.48**	18153	14260	1.27
	pyramid	10392	10472	0.99	13076	11420	**1.15**
	tree	crashed	1244	—	416	1336	**0.31**
double_first	chain	4976	3536	1.41	3636	2520	**1.44**
	cycle	30885	25821	1.20	36122	14976	**2.41**
	grid	3888	2744	**1.42**	2496	1832	1.36
	pyramid	20281	14212	1.43	21353	13428	**1.59**
	tree	784	988	**0.79**	716	1380	0.52
double_last	chain	5100	3528	**1.45**	3680	3048	1.21
	cycle	30601	20293	1.51	30469	15172	**2.01**
	grid	3896	2596	**1.50**	2492	1828	1.36
	pyramid	20201	14196	1.42	17549	10724	**1.64**
	tree	776	972	**0.80**	704	1368	0.51

Table 6.2: Times and speedups for call subsumption in SLG-WAM and YapTab: `path` programs.

- The programs `left_first`, `right_first` and `double_first` with the query '`?- path(X,Y).`'. Note that there is no need to use the `last` versions of these programs, because they will result in the same number of answer trie nodes.

- The `samegen/2` program.

- The `genome/1` program.

Data		**SLG-WAM**			**YapTab**		
		Var	*Sub*	*Speedup*	*Var*	*Sub*	*Speedup*
samegen	chain	23385	28	**835.18**	44	60	0.73
	cycle	6176	12	**514.67**	32	28	1.14
	grid	2228	1352	**1.65**	1436	1068	1.34
	pyramid	2220	20	**111.00**	20	24	0.83
	tree	7132	8828	**0.81**	4532	7620	0.59
genome	chain	9352	16	584.50	19277	20	**963.85**
	cycle	2324	8	290.50	5860	8	**732.50**
	grid	1076	8	**134.50**	1464	16	91.50
	pyramid	1936	4	**484.00**	2240	8	280.00
	tree	41646	28	**1487.36**	43674	48	909.88
reach_first	iproto	2940	2536	**1.16**	1188	1440	0.82
	leader	5844	6040	**0.97**	2176	2420	0.90
	sieve	16637	23413	0.71	14224	15064	**0.94**
reach_last	iproto	2296	2576	**0.89**	1172	1452	0.81
	leader	5796	5992	**0.97**	2160	2412	0.90
	sieve	20653	19309	1.07	16861	14740	**1.14**

Table 6.3: Times and speedups for call subsumption in SLG-WAM and YapTab: `samegen`, `genome`, and model checking programs.

- The `flora` benchmark.

Table 6.4 presents the results we obtained. From the table, we see that, for some programs where no subsumed subgoals are called, the number of answer trie nodes created is exactly same when compared to variant-based tabling. For example, in the programs `genome/1` and `left_first` we can observe this behavior. Please notice that, while the number of trie nodes is the same, the size of the nodes in subsumption-based tabling (they have a **timestamp** field) make the time stamped answer trie more costly in terms of memory used.

In the programs `right_first` and `double_first`, the number of answer trie nodes is reduced in half. Using variant-based tabling, these programs generate a large number of subgoals, that in subsumption-based tabling are consumers of the first called subgoal, thus creating more answer tries for each one of these subgoals, and thus more answer trie nodes. As one would expect, using subsumption-based tabling can reduce substantially the table space.

The `samegen/2` program presents a curious behavior for the `cycle` and `chain` data sets. In these cases, the program generates an answer that subsumes all the other answers and every other subgoal call is subsumed by the top subgoal, resulting in only 3 answer trie nodes created, the root node, and the two nodes for the two terms of the general solution. Compared to tabling with variant checks, this result in very good space savings. The `flora` benchmark also shows a reduced table space size, which shows that subsumption-based tabling can be successfully applied to complex programs, resulting in a largely reduced table space.

Data		**Sub / Var**	**YapTab**	
		ratio	***Variant***	***Subsumptive***
left_first	chain (4096)	1.00000	8390656	8390656
	cycle (4096)	1.00000	16781313	16781313
	grid (64)	1.00000	16781313	16781313
	pyramid (4096)	1.00000	25171968	25171968
	tree (32768)	1.00000	442370	442370
right_first	chain (4096)	0.50012	16777216	8390656
	cycle (4096)	0.50000	33562625	16781313
	grid (64)	0.50000	33562625	16781313
	pyramid (4096)	0.50008	50335744	25171968
	tree (32768)	0.50943	868356	442370
double_first	chain (512)	0.50098	262144	131328
	cycle (512)	0.50000	525313	262657
	grid (16)	0.50000	131585	65793
	pyramid (512)	0.50065	786944	393984
	tree (32768)	0.50943	868356	442370
samegen	chain (32768)	0.00002	131071	3
	cycle (16384)	0.00005	65539	3
	grid (32)	0.49903	1050627	524291
	pyramid (4096)	0.33335	49147	16383
	tree (8192)	0.79965	27974313	22369623
genome	chain (16384)	1.00000	16383	16383
	cycle (8192)	1.00000	8193	8193
	grid (64)	1.00000	4097	4097
	pyramid (4096)	1.00000	4096	4096
	tree (32768)	1.00000	16383	16383
flora		0.36173	83564	30228

Table 6.4: Detailed comparison between variant and subsumption-based tabling for stored answer trie nodes.

6.3 Performance Evaluation for RCS

In this section we evaluate our retroactive-based tabling engine that we implemented on top of YapTab. First, we start by assessing the overhead of using the new mechanisms that support the RCS engine, namely: building the subgoal dependency tree, the STST table space, and searching for running subsumed subgoals. In the second part of this section, we evaluate the RCS engine with programs where specific subgoals are called before general subgoals, in order to assess the advantages of the new mechanism.

6.3.1 RCS Overhead

To measure the overhead, we executed programs where general subgoals are always called before subsumed subgoals (or not at all), therefore we can estimate the impact in the execution time because the pruning techniques of RCS are not employed. We used the following benchmarks:

- Every combination of versions and data sets for the `path/2` program with the query ‘`?- path(X,Y).`’.
- The query ‘`?- samegen(X,Y).`’ in the `samegen/2` program. All `path/2` data sets were used.
- The two versions of the `reach/2` program with the following queries for the relation graphs:
  - sieve: ‘`?- reach(sieve_0(5,4,27,end),Y).`’.
  - leader: ‘`?- reach(systemLeader_0(5,end),Y).`’.
  - iproto: ‘`?- reach(iproto_0(_,_,end),Y).`’.

We timed the execution of the benchmarks for the RCS engine and then the execution time for the subsumptive and variant-based engines of both YapTab and SLG-WAM. In the next tables, we present the execution time of the RCS engine in milliseconds and the relative time of the other engines by computing the value T_{engine}/T_{RCS}. If the value is lesser than 1.0 then RCS performs worse, otherwise if the value is greater than 1.0 then RCS performs better. At the end of each table, we present the average values for each engine.

Tables 6.7, 6.8 and 6.6 show the detailed results for the programs we use to measure the overhead. Table 6.5 shows the average values computed for each benchmark program for call subsumption in the SLG-WAM and in the YapTab engine. By analyzing the results we can see that YapTab with RCS performs worse than YapTab with subsumptive-based tabling in most cases, only the programs `right_first` and the `right_last` program show better results, while the `left_first` program has comparable execution times. In theory, these benchmarks should not run faster, but cache effects and other conditions could affect positively the execution time of these programs.

The average values presented in Table 6.5 show that RCS is very competitive against XSB, because it is, in average, 21% slower than RCS. More importantly, when comparing RCS to YapTab with traditional call subsumption, RCS performs 5% slower, which shows that RCS adds a very small overhead when executing programs that do not benefit from the new evaluation model.

Program	**SLG-WAM**	**YapTab**
	Sub / Retro	***Sub / Retro***
left_first	1.29	**1.00**
left_last	1.26	0.92
right_first	0.92	**1.06**
right_last	1.00	**1.08**
double_first	1.11	0.89
double_last	1.03	0.87
samegen	0.78	0.91
reach_first	1.801	0.93
reach_last	1.703	0.92
Average	1.21	0.95

Table 6.5: Average overhead values for programs do not take advantage of RCS.

6.3.2 RCS Gains

In this section, we present experimental results using retroactive-based tabling on programs that can benefit from it, i.e., programs where some general subgoal is called after a more specific subgoal. The programs we used for these experiments are the following:

Data		Retro	SLG-WAM		YapTab	
		milliseconds	*Var / Retro*	*Sub / Retro*	*Var / Retro*	*Sub / Retro*
reach_first	iproto	1576	1.87	1.61	0.75	0.91
	leader	2644	2.21	2.28	0.82	0.92
	sieve	15496	1.07	1.51	0.92	0.97
reach_last	iproto	1592	1.44	1.62	0.74	0.91
	leader	2680	2.16	2.24	0.81	0.90
	sieve	15388	1.34	1.25	1.10	0.96

Table 6.6: Detailed overhead results: model checking programs.

Data		Retro	SLG-WAM		YapTab	
		milliseconds	*Var / Retro*	*Sub / Retro*	*Var / Retro*	*Sub / Retro*
double_first	chain	2940	1.69	1.20	1.24	0.86
	cycle	20489	1.51	1.26	1.76	0.73
	grid	2108	1.84	1.30	1.18	0.87
	pyramid	12036	1.69	1.18	1.77	1.12
	tree	1568	0.50	0.63	0.46	0.88
double_last	chain	2896	1.76	1.22	1.27	1.05
	cycle	22505	1.36	0.90	1.35	0.67
	grid	2128	1.83	1.22	1.17	0.86
	pyramid	12004	1.68	1.18	1.46	0.89
	tree	1564	0.50	0.62	0.45	0.87
left_first	chain	2544	1.12	1.54	1.02	0.95
	cycle	6876	0.87	1.36	0.98	0.88
	grid	11832	1.14	1.33	0.86	0.96
	pyramid	12672	0.76	1.07	1.01	1.18
	tree	140	0.97	1.14	0.86	1.03
left_last	chain	2512	1.08	1.59	1.04	0.99
	cycle	6996	0.88	1.39	0.98	0.87
	grid	11948	1.12	1.31	0.85	0.96
	pyramid	2780	0.86	1.05	1.04	0.97
	tree	164	0.78	0.95	0.71	0.83

Table 6.7: Detailed overhead results: `double` and `left` programs.

- The programs `left_first`, `left_last`, `double_first` and `double_last` with the query goal '`?- path(X,1).`'. When calling the subgoal `path(X,1)`, the subgoal `path(X,Y)` is called, which prunes the first subgoal.

- The `genome/1` program with the query goal '`?- genome(X).`'.

- The two versions of the `reach/2` program with the following pair of queries/re-

Data		**Retro**	**SLG-WAM**		**YapTab**	
		milliseconds	***Var / Retro***	***Sub / Retro***	***Var / Retro***	***Sub / Retro***
right_first	chain	3160	0.90	0.90	1.06	1.08
	cycle	8148	0.78	0.87	0.82	0.85
	grid	14852	1.22	1.03	1.06	1.22
	pyramid	10836	1.12	0.90	1.09	1.06
	tree	204	0.78	0.90	0.84	1.10
right_last	chain	3152	0.89	0.91	1.17	1.16
	cycle	6196	1.03	1.13	1.21	1.06
	grid	12284	1.61	1.09	1.48	1.16
	pyramid	10952	0.95	0.96	1.19	1.04
	tree	1344	crashed	0.93	0.31	0.99
samegen	chain	72	324.79	0.39	0.61	0.83
	cycle	32	193.00	0.38	1.00	0.88
	grid	1160	1.92	1.17	1.24	0.92
	pyramid	28	79.29	0.71	0.71	0.86
	tree	7124	1.00	1.24	0.64	1.07

Table 6.8: Detailed overhead results: `right` and `samegen` programs.

lation graphs:

- sieve: ‘`?- reach(sieve_0(5,4,27,end), par(A, B, C, D)).`’.
- leader: ‘`?- reach(systemLeader_0(5,end), par(D, E, A, B)).`’.
- iproto: ‘`?- reach(iproto_0(_,_,end),imain_7_0(A, B, C, D, E)).`’.

- The **flora** program with the query goal ‘`?- ´_$_$_flora_isa_rhs(́_,direct).`’.
- The **fib** program with the following parameters: 30, 32, 35, and 36. We use the following query goal: ‘`?- a(X), p(Y, Z).`’.
- The **big** program with the following parameters (number of subgoals to prune): 5, 10, and 20. We use the query goal ‘`?- a(X).`’

As the tables of the previous section, we present the results with the execution time of the retroactive-based engine, along with the relative results for the SLG-WAM and YapTab engines, with both variant and subsumptive checks. In tables 6.12, 6.10 and 6.11 we show the detailed results of the experiments. Table 6.9 presents the average values of each program.

For the `path/2` we should make two distinctions. The versions where the recursive clause is the first clause (`left_first` and `double_first`) and the versions with the

Program	**SLG-WAM**	**YapTab**
	Sub / Retro	***Sub / Retro***
left_first	1.19	0.95
left_last	0.76	0.88
double_first	1.20	**1.13**
double_last	1.14	**1.14**
genome	0.75	0.92
reach_first	3.28	**1.63**
reach_last	3.25	**1.85**
flora	0.29	**1.17**
fib	1.75	**2.19**
big	17.18	**13.60**

Table 6.9: Average benchmarks results for programs that take advantage of RCS.

recursive last as the second clause (`left_last` and `double_last`). In the `first` versions, the specific subgoal generates answers before reaching the general subgoal, while in the `last` versions, the general subgoal is found right at the beginning. Therefore, the `first` versions should, in principle, attain more performance, because they waste less time executing the subsumed subgoal. The results for the `left` versions of the `path/2` program attest this, with speedups of 0.95 and 0.88 for the `left_first` and `left_last` versions, respectively. Surprisingly, the `double` version has the opposite behavior.

Another important conclusion we can make from the results is that RCS may not show performance gains, even if the programs can, in theory, take advantage of it. In our experiments, only the programs `double_left`, `double_last`, `reach_left` and `reach_last` show, in average, performance improvements.

We argue that the speedup of using RCS is highly dependent on the nature of the program. The `left` version of the `path/2` program, for example, does not show improvements because what we gain from pruning the execution of simple `edge/2` facts does not offset the cost of using the STST to retrieve relevant answers for the subsumed subgoal and the cost of pruning the computation itself. In other words, using subsumptive-based tabling for this program is advantageous because the cost of executing predicate clauses is less than maintaining the time stamped index. Therefore, the time that would be wasted by executing pruned code should be bigger than the time wasted on manipulating the STST.

The model checking programs, `reach_left` and `reach_last`, are much more complex than the `path/2` programs, and thus show much larger improvements. For the `reach_last` program with the `leader` model, we obtain a good speedup of 2.03. Also, the `flora` program also shows improvements with a speedup of 1.17 over traditional call subsumption in YapTab. These complex benchmarks show that using RCS has the potential to speedup the execution on this type of programs.

The **fib** program shows a considerable average speedup of 2.19 for YapTab with call subsumption. We can see that, for different parameters, the speedup is maintained because the execution time of the pruned subsumed subgoal is more or less equal to that of the subsuming subgoal. For the **big** program, where multiple subsumed subgoals are pruned, we can see that, as the number of pruned subsumed subgoals increase, the speedup also increases, reaching a maximum of 21.29 for YapTab. An interesting observation for the **big** program is that the execution time for retroactive-tabling stays the same, even if the number of subsumed subgoals to prune increase.

Even if the `path/2` programs, in average, do not show improvements, for some data sets these programs can obtain good speedups. For example, the binary tree configuration can reach speedups of 1.62 and 1.63. The grid configuration can reach speedups of 1.10 for the `left_first` benchmark, and the `genome/1` program reaches a speedup of 2.0 for the grid graph.

Data		**Retro**	**SLG-WAM**		**YapTab**	
		milliseconds	***Var / Retro***	***Sub / Retro***	***Var / Retro***	***Sub / Retro***
reach_first	iproto	2120	2.30	2.46	1.10	1.35
	leader	2644	4.37	4.54	1.84	1.81
	sieve	16801	2.19	2.85	2.12	1.72
reach_last	iproto	1848	2.63	2.85	1.27	1.55
	leader	2772	4.23	4.77	1.59	2.03
	sieve	16053	2.06	2.14	1.76	1.98
flora		72	2.5	0.29	3.17	1.17

Table 6.10: Detailed gain results: model checking programs.

Data		**Retro**	**SLG-WAM**		**YapTab**	
		milliseconds	*Var / Retro*	*Sub / Retro*	*Var / Retro*	*Sub / Retro*
big	5	580	8.27	8.26	5.92	6.17
	10	584	14.88	15.05	10.90	13.35
	20	584	33.76	28.23	20.59	21.29
fib	30	220	4.04	4.00	1.98	2.25
	32	580	crashed	crashed	1.97	1.98
	35	2632	crashed	crashed	1.86	1.86
	36	3916	crashed	crashed	2.02	2.68

Table 6.11: Detailed gain results: `fib` and `big` programs.

Data		**Retro**	**SLG-WAM**		**YapTab**	
		milliseconds	*Var / Retro*	*Sub / Retro*	*Var / Retro*	*Sub / Retro*
left_first	chain	2956	1.03	1.38	0.96	0.90
	cycle	6808	0.92	1.40	0.79	0.77
	grid	13316	1.14	1.31	1.06	1.10
	pyramid	16437	0.75	0.90	0.86	0.94
	tree	184	0.89	0.98	0.89	1.04
left_last	chain	2928	1.23	1.42	0.95	0.90
	cycle	6884	0.92	1.42	0.78	0.79
	grid	16469	1.15	1.08	0.71	0.78
	pyramid	3260	0.82	1.01	0.97	0.91
	tree	184	0.89	0.98	0.85	1.00
double_first	chain	3028	1.87	1.18	1.15	0.93
	cycle	17493	1.76	1.54	1.09	1.16
	grid	2104	1.92	1.24	1.15	0.94
	pyramid	11956	1.74	1.20	1.27	1.01
	tree	1360	0.67	0.82	0.58	1.62
double_last	chain	2952	1.70	1.22	1.19	1.20
	cycle	17205	1.84	1.20	1.29	0.96
	grid	2120	1.83	1.25	1.12	0.93
	pyramid	11992	2.19	1.19	1.29	0.96
	tree	1380	0.65	0.83	0.58	1.63
genome	chain	24	382.00	0.83	802.71	0.50
	cycle	12	192.67	1.00	394.67	0.67
	grid	8	140.00	1.00	181.50	2.00
	pyramid	16	122.25	0.50	158.50	0.50
	tree	64	827.98	0.44	682.66	0.94

Table 6.12: Detailed gain results: `path` and `genome` programs.

6.4 STST Table Space Analysis

In the Single Timed Stamped Trie table space, the answers for all the subgoals of a predicate are stored in a single answer trie. While advantageous, all arguments of the answers must be stored in the trie. In this section, we experiment with programs where this property is stressed to measure the overhead in terms of space and time, when the the load operation is more expensive and the store operation needs to insert more terms in the trie than what is needed to complete the computation.

For this, we used the `path/2` program. We transformed, both the program and data sets, to use a functor term in each argument, instead of simple integers. For example, an `edge(3,4)` fact is transformed into `edge(f(3), f(4))`. The updated version of the `left_first` program is exemplified in Figure 6.1. We experimented the query goal '`?- path(f(X),f(Y))`' with all the combinations of the `path/2` versions and the graph data sets.

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

Figure 6.1: Transformed `path/2` program.

6.4.0.1 Time Analysis

Table 6.14 present the detailed results concerning execution time for each program and data set. The average results are presented in Table 6.13 and compare the STST to tabling with variant and subsumptive checks. From the results, we see that, in average, the STST table space has an overhead of 20%, which is considerable when compared to the overhead of 5% found early on this chapter.

The consumption of answers by consumers and the insertion of new answers by generators into the table space are the primary causes for the overhead in RCS for these benchmarks. The programs with the worst overhead are `double_first` and `double_last`, with 33% and 35% of overhead against traditional call subsumption. These programs also create the higher number of consumers, both variant consumers and subsumed consumers than any other benchmark in this experiment. The `right_first` and `right_last` are the only programs where only subsumed consumers are created, and they have an overhead of 7% and 11%, respectively, which are the lowest overhead values. In the programs `left_first` and `left_last`, only one variant consumer is allocated, however they perform worse than the `right` versions.

We thus argue that the number of consumer nodes can greatly reduce the applicability and performance of the STST table space when the operation of loading an answer from the trie is more expensive. While this situation is disadvantageous, execution time can be reduced when another subgoal call appears (for example `path(X,Y)`), where its possible to reuse the answers from the table instead of executing the predicate clauses, thus providing an elegant solution to the problem of incomplete tables.

Program	**YapTab**	
	Var / Retro	***Sub / Retro***
left_first	0.87	0.86
left_last	0.78	0.77
right_first	0.85	0.93
right_last	0.92	0.89
double_first	0.52	0.67
double_last	0.51	0.65
Average	0.74	0.80

Table 6.13: Average results for the query goal '`?- path(f(X),f(Y))`'.

6.4.0.2 Space Analysis

We executed the previous benchmarks and measured the number of answer trie nodes for each program. The detailed results are presented in Table 6.16. In this table we show the number of answer trie nodes allocated for the RCS execution and then the relative number of trie nodes for the variant and subsumptive-based executions, all for YapTab. The programs `left`, `right` and `double` are the left, right and double versions of the `path/2` program. Note that we do not need to differentiate between `first` and `last` versions, because they generate the same number of trie nodes. For the data sets, we used graph configurations with different sizes.

Table 6.15 presents the average results of RCS against variant and subsumptive-based engines. From these results we can see that the STST table space is 89% more efficient than the variant table space. In the `double` program, these differences are augmented because in the variant engine there are more generator subgoal calls and thus more answer tries are created.

Data		**Retro**	**YapTab**	
		milliseconds	***Var / Retro***	***Sub / Retro***
left_first	chain	792	0.87	0.81
	cycle	1712	0.96	0.79
	grid	16913	0.75	0.99
	pyramid	836	0.89	0.82
	tree	556	0.86	0.88
left_last	chain	1060	0.64	0.61
	cycle	1744	0.93	0.78
	grid	19545	0.64	0.73
	pyramid	828	0.92	0.86
	tree	588	0.79	0.85
right_first	chain	4004	0.82	0.82
	cycle	7300	0.92	1.08
	grid	1140	0.80	0.76
	pyramid	3156	0.86	0.99
	tree	288	0.85	1.00
right_last	chain	3612	1.06	0.92
	cycle	9492	0.80	0.75
	grid	1000	1.18	0.97
	pyramid	3176	0.90	0.90
	tree	532	0.67	0.92
double_first	chain	1696	0.25	0.58
	cycle	2916	0.93	0.54
	grid	3920	0.66	0.69
	pyramid	6804	0.27	0.65
	tree	708	0.50	0.91
double_last	chain	1880	0.23	0.50
	cycle	2924	0.92	0.54
	grid	3928	0.65	0.69
	pyramid	6808	0.27	0.66
	tree	724	0.50	0.87

Table 6.14: Detailed execution time for the query goal '`?- path(f(X),f(Y))`'.

When comparing to the subsumptive-based engine, the STST table space only stores 4.2% more trie nodes, even if the `f/1` functor terms need to be stored. This is easily understandable because the first functor term is only represented once, at the top of the trie, and then there is one second functor for each source node number in the graph, therefore, the number of functors stored is insignificant when compared to the rest of terms that are stored in the trie. Also note that, for the `double` benchmark, the data sets used are small compared to the other data sets used in other benchmarks, and the space overhead is more significant (in the worst case 18%). We thus argue that in the worst case, the extra space needed to store terms in the single answer trie get more insignificant as more terms that are directly related to the query goal are stored in the trie.

Program	**YapTab**	
	Var / Retro	***Sub / Retro***
left	0.99258	0.99258
right	1.95122	0.97906
double	2.72892	0.90149
Average	1.89091	0.95771

Table 6.15: Average space results for the query goal '`?- path(f(X),f(Y))`'.

Data		**Retro**	**YapTab**	
		nodes	***Var / Retro***	***Sub / Retro***
double	chain (256)	26387	2.48365	0.87490
	cycle (256)	36844	3.57141	0.89333
	grid (16)	59028	2.22920	0.82586
	pyramid (256)	56638	3.47583	0.95187
	tree (16384)	213008	1.88449	0.96148
left	chain (2048)	2100233	0.99902	0.99902
	cycle (2048)	4200450	0.99902	0.99902
	grid (64)	16789506	0.99951	0.99951
	pyramid (1024)	1576457	0.99870	0.99870
	tree (65536)	983056	0.96665	0.96665
right	chain (4096)	8398847	1.99756	0.99902
	cycle (4096)	16789506	1.99902	0.99951
	grid (32)	1051650	1.99610	0.99805
	pyramid (2048)	6302719	1.99675	0.99870
	tree (32768)	491520	1.76667	0.90000

Table 6.16: Detailed space results for the query goal '`?- path(f(X),f(Y))`'.

Chapter 7

Conclusions and Further Work

In this chapter we summarize the work developed in this thesis. First, we list the various contributions present in the thesis and then we suggest open problems for future research involving the work developed in this thesis.

7.1 Main Contributions

We can identify two main contributions of this thesis. The first main contribution is the integration of the TST approach for a call subsumption into YapTab's engine. The second main contribution is the design and implementation of a new tabling execution mechanism called Retroactive Call Subsumption, that maximizes the sharing of answers between subsumed/subsuming subgoals. In order to implement the subsumption algorithms and data structures for these two mechanisms we took advantage of the Time-Stamped Tries original proposal from XSB. This process made us to also understand the differences and similarities between the YapTab and the SLG-WAM tabling engines.

We next detail the most important aspects of the work developed during this thesis:

Subsumption-based tabling engine. Tabled evaluation with subsumptive checks is now supported in Yap Prolog. While we integrated the TST algorithms and data structures from SLG-WAM into YapTab, the modifications made to YapTab were minimal and thus show that it is possible to add support for subsumption-based tabling to a delaying-based tabling engine that already supports variant checks by preserving its own tabling operations and main algorithms. In addition, YapTab is now also able to mix variant and subsumption-based tabling in the same program by defining the evaluation model for each predicate.

Mechanisms to control retroactive-based execution. This thesis innovates by presenting a novel execution model that is able to prune the execution of specific subgoals when a more general subgoal appears. We developed rules for pruning a range of choice points and presented the issues that arise and can lead to completion problems when transforming consumer nodes into generators. We also developed an efficient mechanism to detect if a subgoal is internal to the execution of another subgoal by building a subgoal dependency tree.

Algorithm to find subsumed subgoals in the table space. We designed a novel and efficient algorithm that can detect the instances of a subgoal that are currently being evaluated. Our design takes advantage of the existing WAM machinery and data areas. In order to prune the search space, we extended the subgoal trie data structure with information about the status of the subgoals under a subgoal trie node.

Single Time Stamped Trie table space organization. In the STST table space organization we have a single answer trie for each predicate. The answer trie is itself a time stamped trie and permits greater reuse of answers between the subgoals of the same predicate, as answers are represented only once. The design facilitates the pruning of subsumed subgoals because the subgoals can easily identify which answers have already been consumed or generated. We also designed a new optimization, where we throw away the subgoal trie when the most general subgoal completes, therefore saving more memory.

Enhanced answer reuse. The STST table space organization also allows subgoals to immediately reuse answers added by other subgoals. This means that when more general subgoal appears, we can first load the answers already stored by the subsumed subgoals and only then execute the predicate clauses. This is also a very efficient way of handling incomplete tables [Roc07], because we are not only restricted to an incomplete or complete answer set from a single subgoal but we can reuse answers from other subgoals, which can be useful if the subgoal stops execution by the cut operator and is called again later.

Support for mixed tabling checks. Our final system is able to mix variant, subsumption and retroactive-based tabling in the same program. This enables the programmer to choose the best evaluation strategy per predicate, which arguably can augment the power of tabling for real world programming.

Performance Evaluation. We evaluated the performance of our subsumption-based tabling engine against SLG-WAM with very comparable results, which validates our integration efforts. We also observed that by using call subsumption we can potentially cut down on execution time and waste less memory. For retroactive-based tabling, we evaluated the overhead of using the new evaluation mechanism with programs that do not benefit from retroactive evaluation. For programs were retroactive reuse occurs, we validated our approach with good speedups over traditional call subsumption.

7.2 Further Work

Despite the contributions enumerated above, still much work is left to be done in the future. We next summarize further work that can and should be done:

More efficient algorithms in the table space. While the algorithms we used for the STST table space work pretty well for the majority of applications, there are programs were the overhead of various subgoals inserting on the same answer trie can be considerable. Newer algorithms should be developed in order to, while preserving the good results for the majority of the cases, solve these deficiencies. Moreover, the current implementation cannot benefit from a compiled answer trie until the most general subgoal completes. Novel mechanisms must be developed in order to take advantage of the compiled tries optimization while allowing answer insertion at the same time.

Experimentation in real world applications. Our retroactive-based tabling engine still needs more experimentation and testing with real world data and applications in order to refine the implementation. More intensive experimentation may provide a deep analysis on the algorithms implemented and many opportunities to make each algorithm more efficient and/or robust will certainly arise.

Exploration of other execution models. When using retroactive-based tabling, there are some cases where the engine needs to call multiple subgoals of the same predicate in order to calculate the answers for the top subgoal. Sometimes, it would be advantageous to *abstract* the running subgoals into a more general subgoal and then call it. After that, the top subgoal would use the answers generated by the more general subgoal. This mechanism called *call abstraction* was proposed by Johnson et al. [JRRR99] and is based on the idea that it may be useful for some programs to lose goal directness and generate the full set of answers and then select relevant answers from that set. Retroactive-based tabling already includes all the machinery necessary to do that, which makes it a good framework to further support call abstraction by devising various analysis techniques of call patterns.

Appendix A

Benchmark Programs

This appendix describes the programs used to benchmark the tabling engines explored in this thesis. The program rules will be fully presented here and the facts will be shortly described, as they are too big to fit here. The author may be contacted for the full benchmark set.

A.1 Programs

left_first

```
:- table path/2.

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

left_last

```
:- table path/2.

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

right_first

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

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

right_last

```
:- table path/2.

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

double_last

```
:- table path/2.

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

samegen

```
:- table samegen/2.

samegen(X,X).
samegen(X,Y) :- edge(W, X), samegen(W,Z), edge(Z, Y).
```

genome

```
:- table genome/1.
:- table path/2.

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

genome(X) :- path(1, X), path(2, X).
```

reach_first

```
:- table reach/2.

reach(X, Z) :- reach(X, Y), trans(Y, _, Z).
reach(X, Z) :- trans(X, _, Z).
```

reach_last

```
:- table reach/2.

reach(X, Z) :- trans(X, _, Z).
reach(X, Z) :- reach(X, Y), trans(Y, _, Z).
```

flora

```
:- table ’_$_$_flora_fd’/3.
:- table ’_$_$_flora_mvd’/3.
:- table ’_$_$_flora_ifd’/3.
:- table ’_$_$_flora_imvd’/3.
:- table ’_$_$_flora_isa’/2.
:- table ’_$_$_flora_sub’/2.
:- table ’_$_$_flora_fs’/3.
:- table ’_$_$_flora_mvs’/3.
:- table ’_$_$_flora_exists’/1.
:- table ’_$_$_flora_mvd’/2.
:- table ’_$_$_flora_imvd’/2.
:- table ’_$_$_flora_fd_dyn’/3.
:- table ’_$_$_flora_mvd_dyn’/3.
:- table ’_$_$_flora_ifd_dyn’/3.
:- table ’_$_$_flora_imvd_dyn’/3.
:- table ’_$_$_flora_isa_dyn’/2.
:- table ’_$_$_flora_sub_dyn’/2.
:- table ’_$_$_flora_fs_dyn’/3.
:- table ’_$_$_flora_mvs_dyn’/3.
:- table ’_$_$_flora_exists_dyn’/1.
:- table ’_$_$_flora_mvd_dyn’/2.
:- table ’_$_$_flora_imvd_dyn’/2.

’_$_$_flora_fd’(O,M,R)   :- ’_$_$_flora_fd_dyn’(O,M,R).
’_$_$_flora_mvd’(O,M,R)  :- ’_$_$_flora_mvd_dyn’(O,M,R).
’_$_$_flora_ifd’(O,M,R)  :- ’_$_$_flora_ifd_dyn’(O,M,R).
’_$_$_flora_imvd’(O,M,R) :- ’_$_$_flora_imvd_dyn’(O,M,R).
’_$_$_flora_isa’(O1,O2)  :- ’_$_$_flora_isa_dyn’(O1,O2).
’_$_$_flora_sub’(O1,O2)  :- ’_$_$_flora_sub_dyn’(O1,O2).
’_$_$_flora_fs’(O,M,R)   :- ’_$_$_flora_fs_dyn’(O,M,R).
’_$_$_flora_mvs’(O,M,R)  :- ’_$_$_flora_mvs_dyn’(O,M,R).
’_$_$_flora_exists’(O)   :- ’_$_$_flora_exists_dyn’(O).
’_$_$_flora_mvd’(O,M)    :- ’_$_$_flora_mvd_dyn’(O,M).
’_$_$_flora_imvd’(O,M)   :- ’_$_$_flora_imvd_dyn’(O,M).

...
% other predicates
....
```

fib

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(X, V) :-
  X > 1,
  X1 is X - 1,
  X2 is X - 2,
  fib(X1, V1),
  fib(X2, V2),
  V is V1 + V2.

% input parameter
% fib_fact(Number).

do_fib(X) :- fib_fact(T), fib(T, X).

:- table p/2.

a(X) :- p(1,X).

p(1,2).
p(1,X) :- do_fib(X).
```

big

```
fib(0, 1) :- !.
fib(1, 1) :- !.
fib(X, V) :-
  X > 1,
  X1 is X - 1,
  X2 is X - 2,
  fib(X1, V1),
  fib(X2, V2),
  V is V1 + V2.

between_num(Num, Num, Num) :- !.
between_num(Num, Num, Max).
between_num(Num, Min, Max) :-
   Min1 is Min + 1,
   between_num(Num, Min1, Max).

% input parameter:
% big_fact(Number).

b(X) :- big_fact(Max), between_num(X, 1, Max).

:- table p/1.
:- table a/1.

a(0) :- b(X), p(X).
a(0).
a(0) :- p(_).

p(_) :- a(_), fib(32, X).
```

A.2 Facts

chain

A set of graph nodes in a chain configuration. An example with 4 nodes is:

```
edge(1, 2).
edge(2, 3).
edge(3, 4).
```

cycle

A set of graph nodes in a chain configuration, but the last node connects to the first node. An example with four nodes is:

```
edge(1, 2).
edge(2, 3).
edge(3, 4).
edge(4, 1).
```

pyramid

A set of graph nodes forming a pyramid configuration. An example with four nodes (depth 2) is:

```
edge(1,2).
edge(1,3).
edge(2,4).
edge(3,4).
```

tree

A set of graph nodes forming a binary tree configuration. An example with seven nodes (depth 3) is:

```
edge(1,2).
edge(1,3).
edge(2,4).
edge(2,5).
edge(3,6).
edge(3,7).
```

grid

A set of graph nodes forming a grid configuration. An example with two nodes is:

```
edge(1,2).
edge(2,1).
edge(3,4).
edge(4,3).
edge(1,3).
edge(3,1).
edge(2,4).
edge(4,2).
```

leader

Leader election specification defined for 5 processes.

```
% the transition relation graph trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
   (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(medium_0(A,B,C,D),
   in(A,E),medium_0(A,B,[E|C],D)).
...
% auxiliary predicates
...
```

sieve

Sieve specification defined for 5 processes and 4 overflow prime numbers.

```
% the transition relation graph trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
   (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(generator_0(A,B,C,D),out(A,B),D) :-
   E is B+1, not B=<C.
...
% auxiliary predicates
...
```

iproto

Specification for the i-protocol defined for a correct version with a huge window size.

```
fixed(fix).
```

```
% the transition relation graph trans(par(A,end,end,B),nop,B).
trans(par(A,B,C,D),E,par(A,F,G,D)) :-
   (partrans(A,E,B,C,F,G);partrans(A,E,C,B,G,F)).
trans(iproto_0(A,B,C),nop,imain_0(C)).
...
% auxiliary predicates
...
```

References

[AK91] H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction.* The MIT Press, 1991.

[BCR93] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative Commutative Discrimination Nets. In *International Joint Conference on Theory and Practice of Software Development*, number 668 in LNCS, pages 61–74. Springer-Verlag, 1993.

[Car90] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, 1990.

[CR08] J. Costa and R. Rocha. Global Storing Mechanisms for Tabled Evaluation. In *International Conference on Logic Programming*, number 5366 in LNCS, pages 708–712. Springer-Verlag, 2008.

[CW96] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[DRR^{+}95] S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–258, New York, NY, USA, 1995. ACM.

[DS99] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. In *International Workshop on Practical Aspects of Declarative Languages*, number 1551 in LNCS, pages 106–121. Springer-Verlag, 1999.

[Fre62] E. Fredkin. Trie Memory. *Communications of the ACM*, 3:490–499, 1962.

[FSW96] 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.

[GG01] 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.

[GRS91] A. Van Gelder, K. Ross, and J. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.

[Joh00] Ernie Johnson. Interfacing a Tabled-WAM Engine to a Tabling Subsystem Supporting Both Variant and Subsumption Checks. In *Conference on Tabulation in Parsing and Deduction*, 2000.

[JRRR99] E. Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, and P. Rao. A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In *Fuji International Symposium on Functional and Logic Programming*, number 1722 in LNCS, pages 284–300. Springer-Verlag, 1999.

[Kow74] R. Kowalski. Predicate Logic as a Programming Language. In *Information Processing*, pages 569–574. North-Holland, 1974.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[Mic68] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[RFS05] R. Rocha, Nuno A. Fonseca, and V. Santos Costa. On Applying Tabling to Inductive Logic Programming. In *European Conference on Machine Learning*, number 3720 in LNAI, pages 707–714. Springer-Verlag, 2005.

[Roc01] R. Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs.* PhD thesis, Department of Computer Science, University of Porto, 2001.

[Roc06a] R. Rocha. Efficient Support for Incomplete and Complete Tables in the YapTab Tabling System. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 2–17, 2006.

[Roc06b] R. Rocha. Handling Incomplete and Complete Tables in Tabled Logic Programs. In *International Conference on Logic Programming*, number 4079 in LNCS, pages 427–428. Springer-Verlag, 2006.

[Roc07] R. Rocha. On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In *International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 155–169. Springer-Verlag, 2007.

[RRR96] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A Thread in Time Saves Tabling Time. In *Joint International Conference and Symposium on Logic Programming*, pages 112–126. The MIT Press, 1996.

[RRS$^+$95] 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.

[RRS$^+$99] 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.

[RRS$^+$00] C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. Venkatakrishnan. XMC: A Logic-Programming-Based Verification Toolset. In *International Conference on Computer Aided Verification*, number 1855 in LNCS, pages 576–580. Springer-Verlag, 2000.

[RSS00] 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.

[RSS05a] R. Rocha, F. Silva, and V. Santos Costa. Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In *International Conference on Logic Programming*, number 3668 in LNCS, pages 250–264. Springer-Verlag, 2005.

[RSS05b] R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1 & 2):161–205, 2005.

[SDRA] V. Santos Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual*. Available from `http://www.dcc.fc.up.pt/~vsc/Yap`.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, 1994.

[SS98] 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.

[SSW94] 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.

[SSW96] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 274–288. The MIT Press, 1996.

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

[Tar72] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TS86] 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.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[WPP77] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog – The Language and its Implementation Compared with Lisp. *ACM SIGPLAN Notices*, 12(8):109–115, 1977.

[YK00] G. Yang and M. Kifer. Flora: Implementing an Efficient Dood System using a Tabling Logic Engine. In *Computational Logic*, number 1861 in LNCS, pages 1078–1093. Springer-Verlag, 2000.

[ZSYY00] 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.