Cláudio Humberto Caldas da Silva

On Applying Program Transformation to Implement Tabled Evaluation in Prolog

U.PORTO

FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

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

2007

Cláudio Humberto Caldas da Silva

On Applying Program Transformation to Implement Tabled Evaluation in Prolog

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Informática

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2007

To my family

Acknowledgments

I would like to thank and express my sincere gratitude towards my supervisors, Ricardo Rocha and Ricardo Lopes, both professionally and personally, for their advise, motivation and support. Their suggestions, availability, needed criticism and inspiring words were always of great help in development and conclusion of this thesis. Thank you very much for your help.

To David Pereira, Tania Magalhães, Tiago Soares and David Vaz for the excellent environment provided in the office and for all the help during this thesis, you were always available for any doubt.

To my mom and dad for their unconditional love and support. To my brothers and my sisters-in-law for all the affection and understanding. To my aunt for all the affection during these years. To my friends Gilberto and Cunha for the long conversations and great help. To all my friends who somehow I knew.

I would like to express a final acknowledgment to all of those that throughout my life, at different levels and by different means, had contributed to shape my personality and help me to be the person I am. Thank you all!

Cláudio Silva
January 2007

Abstract

Tabling is a technique of resolution that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. We can distinguish two main categories of tabling mechanisms: *delaying-based tabling* and *linear tabling*. Delaying-based tabling mechanisms need to preserve the state of suspended tabled subgoals in order to ensure that all answers are correctly computed. Linear tabling mechanisms maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations.

In this thesis we present the design, implementation and evaluation of three different delaying-based and linear tabling mechanisms to support tabled evaluation in Prolog, and for that we apply source level transformations to a tabled program. The transformed program then uses external tabling primitives that provide direct control over the search strategy. To implement the tabling primitives we took advantage of the C language interface of the Yap Prolog system.

Performance results, on a set of common benchmarks for tabled execution, allows us to make a first and fair comparison between these different tabling mechanisms and, therefore, better understand the advantages and weaknesses of each. Our results show that delaying-based tabling obtains better results than linear tabling and, in particular, for programs with complex dependencies, delaying-based tabling clearly outperforms linear tabling. Our results also show that our delaying-based mechanism is comparable to the state-of-the-art YapTab system, that implements tabling support at the low-level engine. We thus argue that our approach is a good choice to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine.

Resumo

A tabulação é uma das mais bem sucedidas técnicas de implementação que resolve a incapacidade dos sistemas tradicionais de Prolog em lidarem com computações redundantes e ciclos infinitos. A ideia básica da tabulação é guardar as soluções das computações intermédias de modo a que estas possam ser re-utilizadas quando aparecem chamadas repetidas a subgolos tabelados. As estratégias de tabulação dividem-se em duas categorias principais: a *tabulação com espera* e a *tabulação linear*. Para assegurar que todas as soluções são correctamente encontradas, a tabulação com espera utiliza um mecanismo de suspensão no qual preserva o estado das sub-computações correspondentes a subgolos tabelados. A tabulação linear usa a simples execução em árvore onde os subgolos tabelados são re-executados sucessivamente até a computação atingir o seu ponto-fixo.

Nesta tese descreve-se o desenho, implementação e avaliação de três das mais conhecidas estratégias de tabulação com espera e tabulação linear. A implementação de cada uma destas estratégias é conseguida por re-escrita dos programas originais onde se incluem primitivas externas de suporte à execução com tabulação, e para implementar essas primitivas utiliza-se o interface à linguagem C disponível nos sistemas de Prolog.

O estudo realizado permitiu-nos fazer uma primeira e justa comparação entre as três estratégias implementadas de modo a melhor compreender as vantagens e limitações de cada uma. Os resultados mostram que a tabulação com espera obtém uma melhor performance do que a tabulação linear e que, em particular, para programas com dependências complexas, a tabulação com espera é claramente superior. Os resultados obtidos também mostram que a nossa estratégia baseada em espera é comparável com o sistema de tabulação YapTab que implementa suporte para tabulação ao nível da máquina de execução de Prolog. Este resultado é bastante interessante porque permite considerar a nossa aproximação de utilizar re-escrita juntamente com primitivas externas como uma alternativa viável, rápida e simples de incorporar tabulação em sistemas Prolog.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Logic programming languages provide a high-level, declarative approach to programming. Arguably, Prolog is the most popular and powerful logic programming language. The interest in Prolog has increased considerably when in 1983, David H. D. Warren proposed a new abstract machine for executing compiled Prolog code that has come to be known as the Warren Abstract Machine, or simply WAM [War83]. The WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM's technology. The advances made in the compilation technology of sequential implementations of Prolog proved to be highly efficient which has enabled Prolog compilers to execute programs nearly as fast as the conventional programming languages like C [Roy90].

The operational semantics of Prolog is given by SLD resolution [Llo87], a resolution strategy particularly simple that matches current stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. For example, for a recursive definition of the transitive closure of a relation, a subgoal may never terminate if the program contains left-recursion or if the graph represented by the program contains cycles even if no clause is left-recursive. For a natural definition of the Fibonacci function, the evaluation of a subgoal under SLD resolution spawns an exponential number of subgoals, many of which are redundant. This lack of completeness and efficiency in evaluating recursive programs mean that Prolog programmers must be concerned with SLD semantics throughout program development. Ideally, one would want Prolog programs to be written as logical statements first, and for control to be tackled as a separate issue.

Tabling [TS86, CW96] is a technique of resolution that overcomes some limitations of traditional Prolog in dealing with recursion and redundant sub-computations. Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. Tabling based models are able to reduce the search space, avoid looping, and have better termination properties than SLD based models [CW96]. The added power of tabling has proved its viability in application areas such as Deductive Databases [SSW94], Program Analysis [RRS$^+$00], Knowledge Based Systems [YK00], and Inductive Logic Programming [RFS05]. This process of remembering and reusing previously computed answers, also known as *memoization* or *tabulation*, was first used to speed up the evaluation of functions [Mic68]. The tabling concept also forms the basis of a transformation used with bottom-up evaluation to compute answers for deductive database queries, that is known by the generic name of *magic* [BR91].

The idea of tabling for logic programming has been proposed from a number of different starting points and given a number of different names: OLDT [TS86], Extension Tables [Die87] and SLD-AL [Vie89] are the better known. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system [SSW94, RSS$^+$97], the most well-known tabling Prolog system, and in particular in the SLG-WAM [SS98], the most successful engine of XSB. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Tabling mechanisms are now widely available in systems like XSB [SS98, DS98, DS00], YapTab [RSS00], B-Prolog [ZSYY00], ALS-Prolog [GG01], and Mercury [SS06]. In these implementations, we can distinguish two main categories of tabling mechanisms: *delaying-based tabling mechanisms* and *linear tabling mechanisms*.

Delaying-based tabling mechanisms need to preserve the state of suspended tabled subgoals in order to ensure that all answers are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. The SLG-WAM [SS98] and the YapTab model [RSS00] preserve the environment of a suspended computation by freezing the stacks. The Mercury implementation [SS06] and two alternative XSB-based models, the CAT [DS98] and CHAT [DS00] models, copy the execution stacks to separate storage. The CHAT model improves the CAT design by combining ideas from the SLG-WAM with those from the CAT: it avoids copying all the execution stacks that represent the state of a suspended computation by introducing a technique for freezing stacks without using freeze registers.

On the other hand, linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The main idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. The DRA technique [GG01], as implemented in ALS-Prolog, is based on dynamic reordering of alternatives with repeated calls. This technique tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls, the *looping alternatives*. It then uses the looping alternatives to repeatedly recompute them until a fix-point is reached. The key idea of the SLDT strategy [ZSYY00], as implemented in B-Prolog, is to let a tabled subgoal execute from the backtracking point of a former repeated call. When no answers are available, the repeated call takes the remaining clauses from the former call and tries to produce new answers by using them. The call is then repeatedly re-executed, until a fix-point is reached. The weakness of the linear mechanisms is the necessity of re-computation for computing fix-points.

The common approach used in all these proposals to include tabling support into existing Prolog systems is to modify and extend the low-level engine. Although this approach is ideal for run time efficiency, it is not easily portable to other Prolog systems as engine level modifications are rather complex and time consuming and require changing important components of the system such as the compiler, the code generator, and the data structures that support Prolog execution.

A different approach to incorporate tabled evaluation into existing Prolog systems is to apply source level transformations to a tabled program. The transformed program then uses external tabling primitives that provide direct control over the search strategy to implement tabled evaluation. The idea of program transformation coupled with tabling primitives was first explored by Fan and Dietrich [FD92] that implemented a form of linear tabling using source level program transformation and tabling primitives implemented as Prolog built-ins. Ramesh and Chen [RC97] implemented a form of delaying-based tabling supporting SLG resolution [CW96], but they used the C language interface, available in most Prolog systems, to implement the tabling primitives. These approaches may compromise efficiency, if compared to systems that implement tabling support at the low-level engine, but allow tabling to be easily incorporated into other Prolog systems.

1.1 Thesis Purpose

This thesis addresses the design, implementation and evaluation of three different mechanisms to support tabled evaluation in Prolog. We have followed the approach that couples program transformation with tabling primitives. To implement the program transformation step, we have extended the original program transformation module of Ramesh and Chen [RC97] to include the tabling primitives for our mechanisms. According to the tabling mechanism to be used, a tabled logic program is first transformed to include tabling primitives through source level transformations and only then, the resulting program is compiled. No transformation is applied to non-tabled predicates and the performance of Prolog programs without tabling is unaffected. The program transformation module is fully written in Prolog.

To implement the tabling primitives we took advantage of the C language interface of the Yap Prolog system [SDRA] to build external Prolog modules implementing the support for each mechanism. We can distinguish two main components in each module: the component that implements the table space data structures and the component that implements the specific control primitives of each mechanism. To implement the table space we used *tries* as proposed by Ramakrishnan *et al.* [RRS$^+$99]. Tries have proved to be one of the main assets of the XSB system, because they are quite compact for most applications, while having fast look-up and insertion. The table space component is commonly used by all mechanisms.

Our tabling primitives include support for the most successful delaying-based and linear tabling mechanisms. We have implemented support for a delayed-based tabling mechanism based on SLG resolution [CW96], that we named *tabled evaluation with continuation calls*, and for the DRA [GG01] and SLDT [ZSYY00] linear tabling mechanisms. All mechanisms are based on a *local scheduling* strategy [FSW96] and support tabled evaluation for definite programs, that is, for programs without negation. The implementation is independent from the Yap Prolog's engine and both source level transformations and tabling primitives can be easily ported to other Prolog systems with a C language interface.

Performance results, on a set of common benchmarks for tabled execution, allows us to make a first and fair comparison between these different tabling mechanisms and, therefore, better understand the advantages and weaknesses of each. Our results show that globally delaying-based tabling obtains better results than any of the linear tabling mechanisms. In particular, for programs with complex dependencies, delaying-based tabling clearly outperforms linear tabling. For these kind of programs, linear tabling clearly pays the cost of performing re-computation to compute fix-points.

Our results also show that our delaying-based mechanism is comparable to the state-of-the-art YapTab system, that implements tabling support at the low-level engine. This is an interesting result because YapTab also implements a delaying-based mechanism based on SLG resolution, uses tries to implement the table space and is implemented on top of the Yap Prolog system. This is thus a first and fair comparison between the approach of supporting tabling at the low-level engine and the approach of supporting tabling by applying source level transformations coupled with tabling primitives. We thus argue that our approach is a good choice to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine.

1.2 Thesis Outline

We next provide a brief overview of the eight chapters of this thesis.

Chapter 1: Introduction. The current chapter.

Chapter 2: Logic Programming and Tabling. Provides a brief introduction to the main topics enclosed by this thesis. We discuss logic programming and tabling for logic programs.

Chapter 3: External Prolog Modules in Yap. Presents the key tool of our work: the C language interface to the Yap Prolog system. We start by presenting the main features of the Yap Prolog system and then we focus on how to use its C language interface to build external Prolog modules. We then discuss how we use the trie data structure to organize the table space and we describe in detail the module that supports its implementation.

Chapter 4: Tabled Evaluation with Continuation Calls. Describes the design and implementation of the tabled evaluation with continuation calls mechanism. First, we introduce the basic execution model and show how a tabled program is transformed for this mechanism. Next, we present an example that shows the interaction between Prolog execution and the tabling primitives for this mechanism. We then provide the details for implementing this mechanism as an external Prolog module written in C and discuss how completion is detected.

Chapter 5: Dynamic Reordering of Alternatives. Describes the design and implementation of the DRA linear tabling mechanism as proposed by Guo *et al.* [GG01]. Initially, we describe the basic execution model for the DRA technique and then we show how a tabled program is transformed to include specific tabling primitives for this mechanism. Next, we present an example showing the interaction between Prolog execution and the tabling primitives for DRA evaluation. We then provide the details for implementing this mechanism as an external Prolog module in Yap.

Chapter 6: SLDT Linear Tabling. Describes the design and implementation of the SLDT linear tabling mechanism as proposed by Zhou *et al.* [ZSYY00]. We describe the basic execution model for SLDT and present an example showing how a tabled program is transformed to include specific tabling primitives for this mechanism. We then present an evaluation example and provide the details for implementing SLDT evaluation as an external Prolog module in Yap.

Chapter 7: Experimental Results. Presents a detailed performance study of the three tabling mechanisms implemented. We describe the set of tabled benchmark programs used and we present the performance results of our three tabling mechanisms on those programs. We then discuss several statistics gathered during execution so that the performance results, advantages and weaknesses of each tabling mechanism can be better understood.

Chapter 8: Conclusions. Summarizes the work, enumerates the contributions and suggests directions for further work.

Chapter 2

Logic Programming and Tabling

This chapter presents a brief overview of the main topics enclosed by this thesis. We discuss logic programming with a focus on the Prolog language and tabling for logic programs.

2.1 Logic Programming

Logic Programming [Llo87] is a programming paradigm based on a subset of a First Order Logic named Horn Clause Logic. Logic programming is a simple theorem prover that, given a theory (or program) and a query, uses the theory to search for alternative ways to satisfy the query. 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 consists of a collection of Horn clauses. A Horn clause is a clause where at most one of the literals is positive:

$$A \vee \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n \qquad \text{or} \qquad \sim B_1 \vee \sim B_2 \vee \ldots \vee \sim B_n$$

which is equivalent to the logical implication of a conjunction of literals:

$$A \Leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n \qquad \text{or} \qquad \Leftarrow B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

A more usual form of writing Horn clauses is:

$$A \leftarrow B_1, B_2, \ldots, B_n \qquad \text{or} \qquad \leftarrow B_1, B_2, \ldots, B_n$$

The literal A is defined as the *head* of the clause, while the conjunction $B_1, \ldots, B_n$ represents the *body* of the clause. Each B_i is called a *subgoal*. If the head of a clause is empty, then the clause is called a *query*. On the other hand, if the body is empty, then the clause is called a *fact*. If the head and the body are both non-empty, then the clause is called a *rule*. A sequence of clauses with the same functor in the head form a *predicate* or *procedure*. Predicates can be formed with facts and/or rules.

Each literal in a Horn clause has the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol and all t_i's are *terms*. A term is either a *constant* (also called *atom*), a *compound term* or a *variable*. Compound terms are structured data objects of the form $p(t_1, \ldots, t_n)$, where p is a *functor* with arity n and each t_i is also a term. Variables are assumed to be universally quantified. Their main characteristics are:

- Variables are logical variables which can be 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.

The computation process is mainly based on two mechanisms: *resolution* and *unification* [Rob65]. There are several strategies for the resolution mechanism. Prolog, for example, uses a top-down resolution mechanism known as *SLD resolution* [Llo87]. In this form of resolution, the first subgoal in a query is matched against a clause generating a new query called the *resolvent*. The resolvent is formed by the body of the matching subgoal and by the remainder subgoals in the initial query. This process is recursively applied until either a subgoal fails at finding a matching clause, or until an empty query is generated. At unification failure, the execution backtracks and tries to find another form to satisfy the original query.

2.1.2 The Prolog Language

Arguably, Prolog is the most popular logic programming language. Prolog was made a viable language when in 1977 David Warren developed the first compiler for Prolog [War77]. This system showed good performance, comparable to the best Lisp implementations of that time [WPP77]. Later, Warren proposed a new abstract machine for executing compiled Prolog code known as the *Warren Abstract Machine*, or simply *WAM* [War83]. The WAM became the most popular way of implementing Prolog and almost all current Prolog systems are based on WAM's technology. The advances made in the technology of sequential compilation of implementations of Prolog in the last two decades, allow state-of-the-art Prolog systems to be highly efficient with comparative performance to imperative languages such as C [Roy90].

Prolog systems use SLD resolution with a fixed selection function: the leftmost subgoal is always selected first. If several alternatives for the subgoal are available, Prolog systems use the order of the clauses in the program. When the computation fails, the execution backtracks to the previous state where it had left unexplored alternatives. The search tree is thus explored from top to bottom and in a left to right manner. To better illustrate how Prolog works, we next present in Fig. 2.1 a small Prolog program that implements the well-known Fibonacci function.

```
fib(1,1).
fib(2,1).
fib(N,R):- N>2,
           N1 is N-1,
           N2 is N-2,
           fib(N1,R1),
           fib(N2,R2),
           R is R1+R2.
```

Figure 2.1: The Fibonacci function

The program includes 3 clauses that form the `fib/2` predicate. The first argument of `fib/2` is the input argument for the Fibonacci function, while the second is the output argument that computes the result. The first and second clauses simply state that the Fibonacci of 1 and 2 (input arguments) is 1 (output argument). The third clause is the recursive rule that computes the Fibonacci function. Initially, it checks if the input argument `N` is greater than 2 and, when this is the case, it calls recursively itself twice with the first argument set to `N-1` and `N-2`. The final result is the sum of the results obtained in these two calls.

Note that the head and the body of a rule are separated by the symbol `':-'` (read as *if*) and the subgoals in the body of a rule are separated by the symbol `','` (read as *and*). Variable names start with capital letters and names for constants start with lower case letters. We next show in Fig. 2.2 the execution sequence for the query goal `fib(4,R)`.

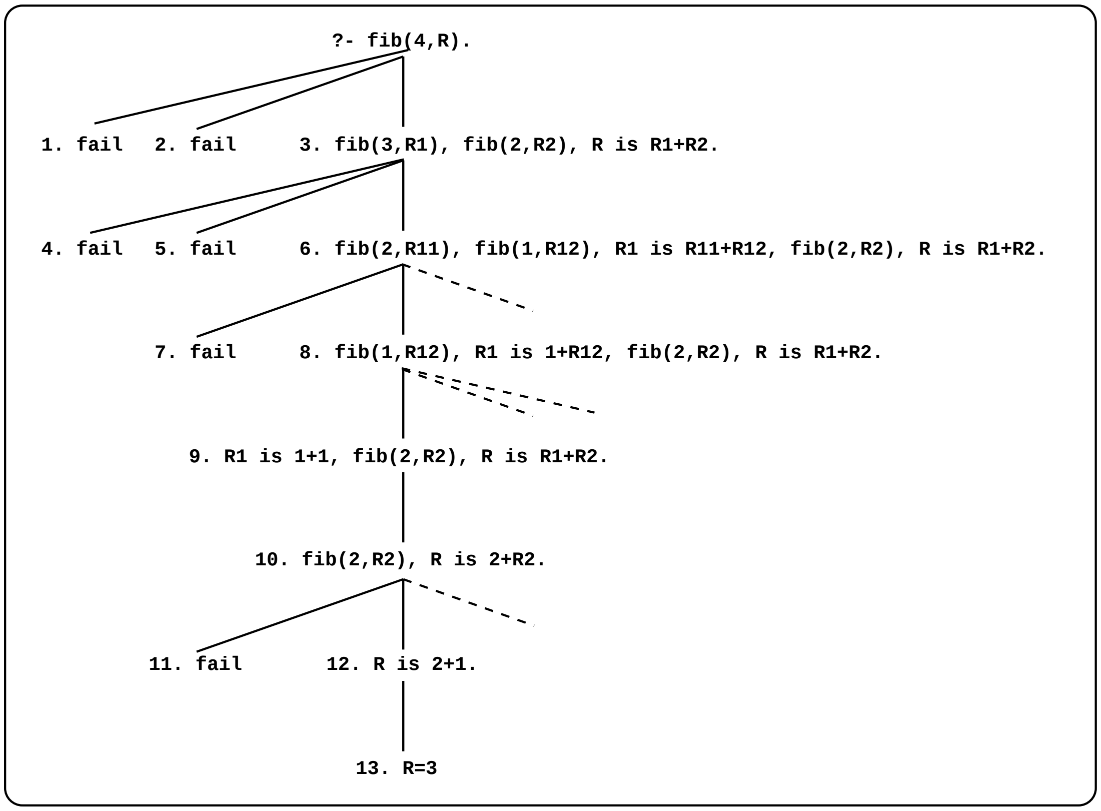

?- fib(4,R).
1. fail
2. fail
3. fib(3,R1), fib(2,R2), R is R1+R2.
4. fail
5. fail
6. fib(2,R11), fib(1,R12), R1 is R11+R12, fib(2,R2), R is R1+R2.
7. fail
8. fib(1,R12), R1 is 1+R12, fib(2,R2), R is R1+R2.
9. R1 is 1+1, fib(2,R2), R is R1+R2.
10. fib(2,R2), R is 2+R2.
11. fail
12. R is 2+1.
13. R=3

Figure 2.2: The execution tree for the query goal `fib(4,R)`

In order to make Prolog a practical language capable of solving real-world problems, Prolog systems have several control operators and built-in predicates that are not in the first-class logic. The most relevant built-in predicates are:

Meta-logical predicates: inquire the state of the computation and manipulate terms. The most well-known are the `var/1` and `atom/1` family of built-in predicates that test the state of the arguments.

Control predicates: perform control operations. *Cut* is the most popular control operator in Prolog programs. When executed, cut discards all alternatives created since the start of the current procedure.

Extra-logical predicates: manipulate the Prolog database by adding or removing clauses from the program being executed and perform input/output operations. The most important are `assert/1`, that adds a clause to the Prolog database, `retract/1`, that deletes a clause from the Prolog database, and `read/1` and `write/1` that implement the basic input and output operations.

Other predicates: these include built-ins to perform arithmetic operations, to compare terms, to obtain the complete set of answers for a query, to support debugging and to interact with the program's environment.

For a more detailed presentation of the Prolog language please refer to some of the standard textbooks on Prolog, such as [CM94, Llo87, SS94].

2.2 Tabling for Logic Programs

Despite the power, flexibility and good performance that Prolog has achieved, the past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. A major problem with Prolog is that SLD resolution presents some fundamental limitations when dealing with recursion and redundant sub-computations. The limitations of SLD resolution mean that Prolog programmers must be concerned with SLD semantics throughout program development. For instance, it is in fact quite possible that logically correct programs will enter infinite loops. Consider, for example, the Prolog program of Fig. 2.3 that defines a small directed graph, represented by the `edge/2` predicate, with a relation of reachability given by the `path/2` predicate, and the query goal `path(1,Z)`. By using SLD resolution to solve the given query leads us to an infinite loop because the first clause of `path/2` recursively calls `path(1,Z)`.

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

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

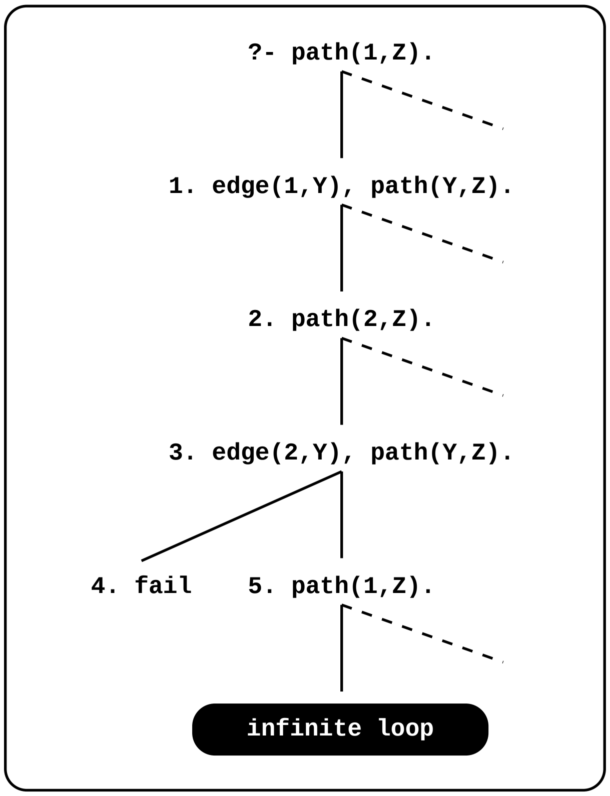


Figure 2.3: An infinite SLD evaluation

A proposal that overcomes some of the limitations of SLD resolution and therefore improves the declarativeness and expressiveness of Prolog is the use of *tabling* [TS86, CW96]. In a nutshell, tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated call appears during the resolution process. Resolution strategies based on tabling are able to reduce the search space, avoid looping, and have better termination properties than traditional Prolog models based on SLD resolution [CW96].

2.2.1 Tabled Evaluation

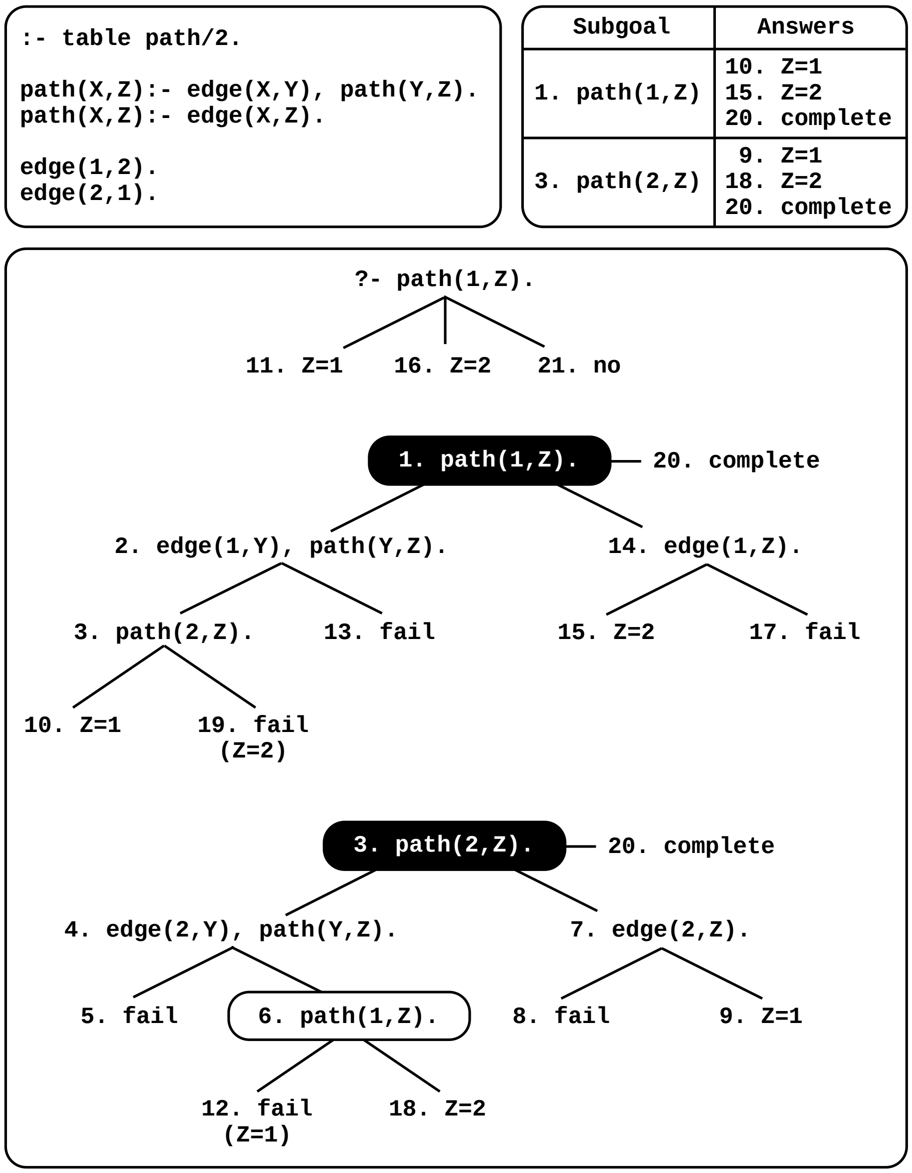
The basic idea behind tabling evaluation is straightforward: whenever a tabled subgoal is first called, a new entry is allocated in an appropriated data space called the *table* *space*. Table entries are used to verify whether calls to subgoals are repeated[1] and to collect the answers found for their corresponding subgoals. Repeated calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

Figure 2.4 uses the same program and the same query goal of Fig. 2.3 to illustrate the main principles of tabled evaluation. At the top, the figure shows the program code (the left box) and the final state of the table space (the right box). Declaration '`:-table path/2.`' in the program code indicates that calls to predicate `path/2` should be tabled. The sub-figure below shows the evaluation sequence for the query goal `path(1,Z)`. Generator nodes are depicted by black oval boxes, and consumer nodes by white oval boxes. Remember that SLD resolution enters an infinite loop because the first clause of `path/2` leads to a repeated call to `path(1,Z)`. In contrast, as we will see, with tabled evaluation termination is ensured.

The evaluation starts by adding a new entry to the table space and by allocating a generator node to represent `path(1,Z)`. Next, `path(1,Z)` is resolved against the first clause for `path/2`, calling `edge(1,Y)` (step 2). The `edge/2` predicate is then resolved as usual because it is not tabled. The first clause for `edge(1,Y)` succeeds with `Y=2`, and in the continuation we call `path(2,Z)` (step 3). As this is the first call to `path(2,Z)`, we add a new entry to the table, and proceed by allocating a new generator node as shown in the bottommost tree. Again, `path(2,Z)` is resolved against the first clause for `path/2`, calling `edge(2,Z)` (step 4). The first clause for `edge(2,Z)` fails, but the second succeeds creating consumer node 6. Since `path(1,Z)` is a repeated call to the initial subgoal, no new tree is created, and instead, we try to consume answers from the table. At this point, the table does not have answers for `path(1,Z)`, and thus, the current evaluation is *suspended*. Consumers must suspend because new answers may still be found to the corresponding call.

The only possible move after suspending is to backtrack to node 3. We then try the second clause for `path(2,Z)`, thus calling `edge(2,Z)` (step 7). The first clause for `edge(2,Z)` fails, but the second succeeds obtaining a first answer for `path(2,Z)` (step

[1]We say that a subgoal repeats a previous subgoal if they are the same up to variable renaming.


```
:- table path/2.

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

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

Subgoal	Answers
1. path(1,Z)	10. Z=1 15. Z=2 20. complete
3. path(2,Z)	9. Z=1 18. Z=2 20. complete

Figure 2.4: A finite tabled evaluation

9). We then follow a Prolog-like strategy and continue forward execution. We thus return to the context of `path(1,Z)` and the binding `Z=1` is propagated to it, also obtaining a first answer for `path(1,Z)` (step 10) and a first solution for the query goal (step 11).

We then return to node 3, but now it has no more clauses left to try. So, we check whether consumer node 6 can be resumed. It can, as now it has unconsumed answers. We thus resume the computation at node 6 and forwarded the answer `Z=1` to it. The subgoal succeeds trivially with a new answer for `path(2,Z)` (step 12). However, this answer repeats what we have found in step 9. Tabled resolution does not store duplicate answers in the table. Instead, repeated answers *fail*. This is how we avoid unnecessary computations, and even looping in some cases.

At this point, all answers have been consumed and, thus, the current evaluation is suspended again. Execution thus backtracks to node 3 and we check whether `path(2,Z)` can be *completed*. It can not, because it depends on subgoal `path(1,Z)` (node 6). Completing `path(2,Z)` earlier is not safe because, at this point, new answers can still be found for subgoal `path(1,Z)`. If new answers are found, node 6 should be resumed with the newly found answers, which in turn can lead to further answers for subgoal `path(2,Z)`. If we complete sooner, we can lose such answers.

Execution thus backtracks to node 2, fails in step 13 and returns to node 1. We then try the second clause for `path(1,Z)`, thus calling `edge(1,Z)` (step 14). The first clause for `edge(1,Z)` succeeds with `Z=2` obtaining a new answer for `path(1,Z)` (step 15) and a new solution for the query goal (step 16). In the continuation, the second clause for `edge(1,Z)` fails (step 17) and backtracking sends us back to node 1. Node 1 has no more clauses left to try, so we check whether consumer node 6 can be resumed. It can, as it has new unconsumed answers. We thus forwarded the new answer `Z=2` to it. This gives new answers to `path(1,Z)` (step 18) and to `path(1,Z)` (step 19). However, this last answer repeats what we have found in step 10, so we fail and backtrack again to node 1. The trees for subgoals `path(1,Z)` and `path(2,Z)` are now fully exploited. As these subgoals do not depend on any other subgoal, we are sure no more answers are forthcoming. So, at last, we declare the two subgoals to be *completed* (step 20) and return *no* to the query goal (step 21).

One of the major characteristics of this execution model is that it can ensure termination for a wider class of programs. This can be useful when dealing with applications with recursive predicates, such as the `path/2` predicate, that can lead to infinite loops. Moreover, as tabling based models are able to avoid re-computation of tabled subgoals, they can reduce the search space and the complexity of a program. This latter property can be explored as a mean to speedup the execution. Consider again the Fibonacci program defined in Fig. 2.1. If predicate `fib/2` is declared as tabled, each different subgoal call is only computed once, as for repeated calls the corresponding answer is already stored in the table space. To compute `fib(n,R)` for some integer `n`, SLD resolution will search a tree whose size is exponential in `n`. Because tabling remembers sub-computations, the number of resolution steps for this example is linear in `n`.

2.2.2 Tabling Operations

The example of Fig. 2.4 show us the four main types of operations required to support tabled evaluation:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if a subgoal is in the table, and if not, adds a new entry for it and allocates a new generator node (nodes 1 and 3 in Fig. 2.4). Otherwise, it allocates a consumer node and starts consuming the available answers (node 6 in Fig. 2.4).

2. The *new answer* operation verifies whether a newly found answer is already in the table, and if not, inserts the answer (steps 9, 10, 15 and 18 in Fig. 2.4). Otherwise, the operation fails (steps 12 and 19 in Fig. 2.4).

3. The *answer resolution* operation forwards answers from the table to a consumer node. It verifies whether newly found answers are available for a particular consumer node and, if any, consumes the next one. Answers are consumed in the same order they are inserted in the table. If no unconsumed answers are available, it *suspends* the current computation and schedules a backtracking node to continue the execution.

4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. A table is said to be *complete* when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program for the subgoal call associated with the table. Otherwise, it is said to be *incomplete*. A table for a tabled subgoal is thus marked as complete when, during evaluation, it is determined that all possible resolutions have been made and, therefore, no more answers can be found (step 20 in Fig. 2.4).

Completion is hard because a number of subgoals may be mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*) [Tar72]. Clearly, we can only complete the subgoals in a SCC together. We will usually represent a SCC through the *leader node*. More precisely, the youngest generator node which does not depend on older generators is called the leader node. For example, in Fig. 2.4, the leader node for the SCC that includes the subgoals `path(1,Z)` and `path(2,Z)` is node 1. A leader node is also the oldest generator node for its SCC, and defines the next completion point.

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

2.2.3 Scheduling Strategies

It should be clear that at several points we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The decision on which operation to perform is crucial to system performance and is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *batched scheduling* and *local scheduling* [FSW96].

Batched scheduling is the strategy we followed in Fig. 2.4: it favors forward execution first, backtracking next, and consuming answers or completion last. It schedules the program clauses in a depth-first manner as does the WAM. It thus tries to delay the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC and delaying the completion point to an older generator node. When backtracking we may encounter three situations: **(i)** if backtracking to a generator or interior node, we try the next clause; **(ii)** if backtracking to a consumer node, we try the next unconsumed answer; **(iii)** if there are no more clauses left to try or no more unconsumed answers, we simply backtrack to the previous node on the current branch. However, if the node without clauses is a leader node, then we first check for completion.

Local scheduling is an alternative tabling scheduling strategy that tries to evaluate subgoals as independently as possible. In this strategy, evaluation is done one SCC at a time. The key idea is that whenever new answers are found, they are added to the table space as usual but execution *fails*. Thus, execution explores the whole SCC before returning answers outside the SCC. Hence, answers are only returned when all program clauses for the subgoal in hand were resolved. Because local scheduling completes subgoals sooner, we can expect less dependencies between subgoals.

Chapter 3

External Prolog Modules in Yap

This chapter presents the main tool of our work - the C language interface to the Yap Prolog system [SDRA]. We start by presenting the main features of the Yap Prolog system and then we focus on how to use its C language interface to build external Prolog modules. Finally, we discuss how we use the *trie* data structure to organize the table space and we describe in detail the module that supports its implementation.

3.1 The Yap Prolog System

Yap Prolog is a high-performance Prolog compiler that extends the WAM with several optimizations for better performance. Yap has been developed since 1985, and its current version is largely compatible with the ISO-Prolog standard. The original version was written in assembly, C and Prolog. Work on the current version of Yap strives at several goals:

Portability: the whole system is now written in C. Yap compiles in popular 32 and 64 bit machines, such as PCs, Suns, Alphas and PowerPCs.

Performance: the Yap emulator is comparable to or better than well-known Prolog systems. The current version of Yap performs better than the original one written in assembly.

Extensibility: Yap was designed internally from the beginning to encapsulate manipulation of terms. These principles were used, for example, to implement a simple and powerful C interface. The new version of Yap extends these principles to accommodate extensions to the unification algorithm, that are useful to implement extensions such as constraint programming.

Completeness: Yap has for a long time provided built-in implementations such as I/O functionality, data-base operations, and modules. Work on Yap aims now at being fully compatible with the Prolog standard.

Robustness: Yap has been tested with a large number of different Prolog applications.

Openness: Yap has been a vehicle for further research and all new developments of Yap are open to the user community. The recent version of Yap is distributed under two licenses: the Free Software Foundation LGPL or the Perl Artistic license 2. The sources to the system are always available from the home page, and contributions from users are always welcome.

3.2 The C Language Interface to Yap

As many other Prolog systems, Yap has an interface for writing predicates in other languages, such as C, as external modules. Figure 3.1 presents a small example for a module written in C, `my_rand.c`, that illustrates how external modules work in Yap. The module defines a new predicate, `my_random/1`, that allows to generate random numbers, that is, a call to `my_random(N)` will unify `N` with a random number.

```
#include "Yap/YapInterface.h"      // header file for the Yap interface to C

void init_predicates(void) {
  Yap_UserCPredicate("my_random", c_my_random, 1);
}

int c_my_random(void) {
  Yap_Term number = Yap_MkIntTerm(rand());
  return(Yap_Unify(Yap_ARG1,number));
}
```

Figure 3.1: The `my_rand.c` module

External modules should be compiled to a shared object and then loaded under Yap by calling the `load_foreign_files()` predicate. After loading a module, Yap initializes it by calling the `init_predicates()` procedure. The `Yap_UserCPredicate()` function is then used to tell Yap what predicates are being defined in the module. In our example, the `my_random/1` predicate is implemented by the `c_my_random()` function. The third argument of `Yap_UserCPredicate()` defines the arity for the predicate. The `c_my_random()` function uses one local variable of the type `YAP_Term`, the type used to hold Yap terms, to store the integer returned by the standard Unix function `rand()`. The conversion is done by `YAP_MkIntTerm()`. Next, it calls `YAP_Unify()`, to attempt the unification with `YAP_ARG1`, and returns an integer denoting success or failure. The arguments of a Prolog predicate written in C are accessed through the macros `YAP_ARG1`, ..., `YAP_ARG16` or with `Yap_A(N)` where `N` is the argument number.

3.2.1 Yap Terms

Terms in Yap can be classified as [SDRA]:

- *variables*;
- *integers*;
- *floating-point numbers* (or *floats*);
- *symbolic constants* (or *atoms*);
- *pairs*;
- *compound terms.*

Integers, floats and atoms are respectively denoted by the following primitives: `Yap_Int`, `Yap_Float` and `Yap_Atom`. For atoms, Yap includes primitives for associating atoms with their names: `Yap_AtomName()` returns a pointer to the string for the atom; and `Yap_LookupAtom()` looks up if an atom is in the standard hash table, and if not, inserts it. A pair consists of two terms, the *head* and the *tail* of the term. Pairs are used to represent lists. One pair can be created using `YAP_MkPairTerm()` where it is necessary to identify the head and the tail of the term. Another alternative to create one pair is to use `YAP_MkNewPairTerm()` where the head and tail are new unbound variables. By using the `YAP_HeadOfTerm()` and `YAP_TailOfTerm()` primitives, it is possible to fetch the head and the tail of a pair.

A compound term is represented by a *functor* and a sequence of terms with length equal to the *arity* of the functor. A functor, identified in C by `YAP_Functor`, consists of an atom (functor name) and an integer (functor arity). As for pairs, compound terms can be constructed from a functor and an array of terms using the

`YAP_MkApplTerm()` primitive. If the array of the terms are unbound variables then one may use the `YAP_MkNewApplTerm()` primitive. Yap also includes the following primitives: `YAP_MkFunctor()` to create functors; `YAP_NameOfFunctor()` to obtain the name of a functor; `YAP_ArityOfFunctor()` to get the arity of a functor; `YAP_FunctorOfTerm()` to fetch the functor of a compound term; and `YAP_ArgOfTerm()` to fetch the term corresponding to a given argument of a compound term. Table 3.1 presents the complete set of the available primitives to test, construct and destruct Yap terms.

Term	**Test**	**Construct**	**Destruct**

variable	Yap_IsVarTerm()
	Yap_NonVarTerm()
		Yap_MkVarTerm()	(none)

integer	Yap_IsIntTerm()	Yap_MkIntTerm()	Yap_IntOfTerm()
float	Yap_IsFloatTerm()	Yap_MkFloatTerm()	Yap_FloatOfTerm()

atom	Yap_IsAtomTerm()	Yap_MkAtomTerm()
		Yap_LookupAtom()
			Yap_AtomOfTerm()
			Yap_AtomName()

pair	Yap_IsPairTerm()	Yap_MkNewPairTerm()
		Yap_MkPairTerm()
			Yap_HeadOfTerm()
			Yap_TailOfTerm()

compound term	Yap_IsApplTerm()	Yap_MkNewApplTerm()
		Yap_MkApplTerm()
			Yap_ArgOfTerm()
			Yap_FunctorOfTerm()

		Yap_MkFunctor()	Yap_NameOfFunctor()
			Yap_ArityOfFunctor()

Table 3.1: Primitives for manipulating Yap terms

We next present in Fig. 3.2 an example that illustrates how these primitives can be used to construct and unify Prolog terms. Consider that we want to construct two compound terms, `p(VAR1,1)` and `p(a,VAR2)`, and that we want to unify variable `VAR1` with atom `a` and variable `VAR2` with integer `1`. To unify two terms, Yap provides the `Yap_Unify(Yap_Term a,Yap_Term b)` primitive. This primitive returns `TRUE` if the unification succeeds, or `FALSE` otherwise.

3.2.2 Writing Predicates in C

External modules may be used to accomplished two different functionalities. One is to call the Prolog interpreter from C. To do so, one needs to construct a goal `G` and then execute `YAP_CallProlog(G)`. The result will be `TRUE`, if the goal succeeds, or `FALSE` otherwise. When it succeeds, the variables in `G` will store the values they have been unified with. The other interesting functionality is to define predicates. Yap allows two kinds of predicates:

```
YAP_Term arg[2], p1, p2;
YAP_Functor f;
f = YAP_MkFunctor(YAP_LookupAtom("p"),2);           // construct functor p/2
arg[0] = Yap_MkVarTerm();
arg[1] = Yap_MkIntTerm(1);
p1 = YAP_MkApplTerm(f,2,args);          // construct compound term p(VAR1,1)
arg[0] = YAP_MkAtomTerm(YAP_LookupAtom("a"));
arg[1] = Yap_MkVarTerm();
p2 = YAP_MkApplTerm(f,2,args);          // construct compound term p(a,VAR2)
YAP_Unify(t1,t2);                                        // unify both terms
```

Figure 3.2: Constructing and unifying compound terms

Deterministic predicates: which either fail or succeed but are not backtrackable;

Backtrackable predicates: which can succeed more than once.

Deterministic predicates are implemented as C functions with no arguments which should return zero when the predicate fails or a non-zero value otherwise. They are declared with a call to `Yap_UserCPredicate(char *name,int *f(),int arity)`, where the first argument is the name of the predicate, the second the name of the C function implementing the predicate, and the third is the arity of the predicate. The `my_random/1` predicate defined in Fig. 3.1 is an example of a deterministic predicate.

Backtrackable predicates are declared similarly, but using instead two C functions: one to be executed when the predicate is first called, and other to be executed on backtracking to provide (possibly) other solutions. They are declared with a call to `Yap_UserBackCPredicate(char *name,int *f_init(),int *f_cont(),int arity, int sizeof)`, where `name` is the name of the predicate, `f_init()` and `f_cont()` are the C functions used to start and continue the execution of the predicate, `arity` is the arity of the predicate, and `sizeof` is the size of the data to be preserved in the stack when backtracking (its use is detailed next). When returning the last solution from a backtrackable predicate, one should use `YAP_cut_succeed()` to denote success or `YAP_cut_fail()` to denote failure. The reason for using these functions is to avoid `f_cont()` to be called indefinitely when backtracking occurs.

Figure 3.3 shows an example that illustrates how backtrackable predicates can be used. The example defines a new predicate, `less_than/2`, that returns by backtracking in the second argument all the positive integers less than the first argument. For instance, if we call `less_than(10,N)`, the predicate should succeed and provide by backtracking all the positive integers less than 10 for the first 9 calls and fail for the 10th call.

```
void init_predicates(void) {
  YAP_UserBackCPredicate("less_than",c_lt_init,c_lt_cont,2,sizeof(int));
}

int c_lt_init(void) {   // to be executed when the predicate is first called
  int limit, *number;
  YAP_Term n = YAP_ARG1;
  YAP_Term m = YAP_ARG2;

  if (YAP_IsIntTerm(n) && YAP_IsVarTerm(m)) {
    limit = YAP_IntOfTerm(n);
    if (limit > 0) {
      YAP_PRESERVE_DATA(number,int);
      *number = 1;
      return (YAP_Unify(m,YAP_MkIntTerm(*number)));
    }
  }
  YAP_cut_fail();
  return FALSE;
}

int c_lt_cont(void) {                      // to be executed on backtracking
  int limit, *number;
  YAP_Term n = YAP_ARG1;
  YAP_Term m = YAP_ARG2;

  limit = YAP_IntOfTerm(n);
  YAP_PRESERVED_DATA(number,int);
  *number++;
  if (*number < limit) {
    return (YAP_Unify(m,YAP_MkIntTerm(*number)));
  YAP_cut_fail();
  return FALSE;
}
```

Figure 3.3: The `less_than/2` predicate

According to definition of the `less_than/2` predicate, the `c_lt_init()` function is executed first. Initially, it checks if the given arguments are of the desired type, and if they are not, it calls `YAP_cut_fail()` and fails by returning `FALSE`. Otherwise, it converts the first argument to an integer using `YAP_IntOfTerm()` and tests if it is a positive number. If not, it also calls `YAP_cut_fail()` and fails. Otherwise, the argument is unified with 1 using `YAP_Unify()`. The conversion of integer 1 to a Yap term is made by `YAP_MkIntTerm()`. To later obtain by backtracking the next integer, we need to preserve the last returned integer. This is done by calling `YAP_PRESERVE_DATA()` to associate and allocate the memory space that will hold the information to be preserved across backtracking, and by calling `YAP_PRESERVED_DATA()` to get access to it later.

The `c_lt_cont()` function is thus executed when backtracking occurs. Initially, it uses the `YAP_PRESERVED_DATA()` to access the last returned integer, its value is then incremented and when the limit is reached, the function fails as in `c_lt_init()`.

For a more exhaustive description on how to use the C language interface of Yap please refer to its manual [SDRA].

3.3 The Table Space Module

Next we describe the key module of our work, the module that implements the table space. This module is very important since the correct design of the algorithms to access and manipulate the table data is critical to achieve an efficient implementation. Our implementation uses tries as proposed by Ramakrishnan *et al.* [RRS+99]. In what follows, we first briefly discuss how tries work and then we present the implementation details for the module. The table space module is commonly used by all the different tabling mechanisms described towards this thesis.

3.3.1 The Trie Data Structure

Tries were first proposed by Fredkin [Fre62], the name coming from the central letters of the word *retrieval*. Tries were originally invented to index dictionaries, and has since been generalised to index recursive data structures such as terms. Please refer to [Ohl90, McC92, BCR93, Gra96, RRS+99] for the use of tries in automated theorem proving, term rewriting and tabled logic programs. An essential property of the trie structure is that common prefixes are represented only once. The efficiency and memory consumption of a particular trie largely depends on the percentage of terms that have common prefixes. For tabled logic programs, we often can take advantage of this property.

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. At the entry point we have the root node. Internal nodes represent symbols in terms and leaf nodes specify the end of terms. Each root-to-leaf path represents a term described by the symbols labeling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing symbol. When inserting a new term, we start traversing the trie starting at the root node. Each child node specifies the next symbol to be inspected in the input term.

A transition is taken if the symbol in the input term at a given position matches a symbol on a child node. Otherwise, a new child node representing the current symbol is added and an outgoing transition from the current node is made to point to the new child node. On reaching the last symbol in the input term, we reach a leaf node in the trie.

Figure 3.4 presents an example for a trie with three terms. Initially, the trie contains the root node only. Next, we insert $f(X, a)$. As a result, we create three nodes: one for the functor $f/2$, next for the variable X, and last for the constant a (Figure 3.4(a)). The second step is to insert $g(X, b, Y)$. The two terms differ on the main functor, so tries bring no benefit here (Figure 3.4(b)). In the last step, we insert $f(Y, 1)$ and we save the two nodes common with term $f(X, a)$ (Figure 3.4(c)).

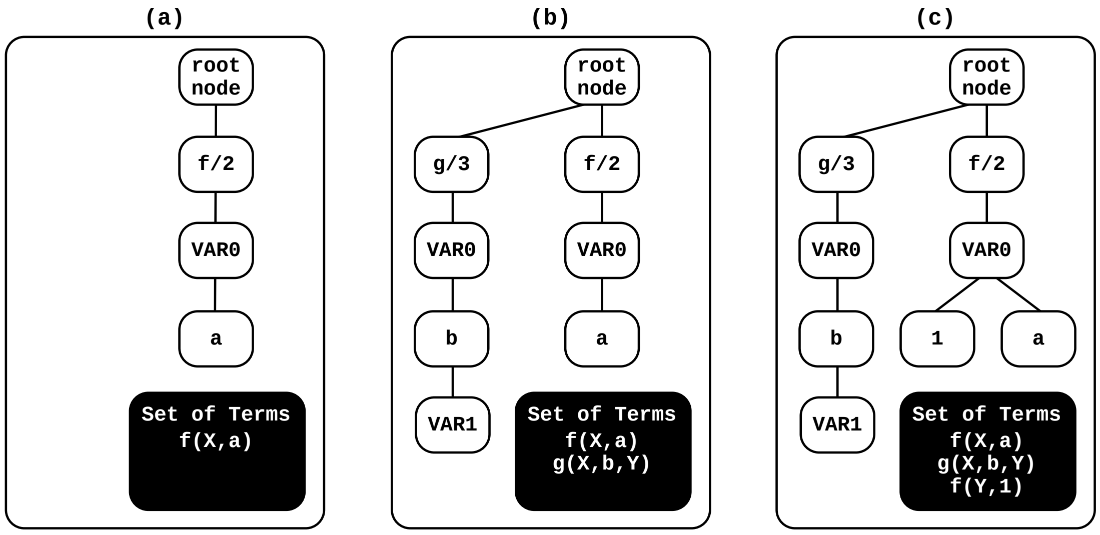


Figure 3.4: Using tries to represent terms

An important point when using tries to represent terms is the treatment of variables. We follow the formalism proposed by Bachmair *et al.* [BCR93], where each variable in a term is represent as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR0, ..., VARN$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t. For example, in the term $g(X, b, Y)$, $numbervar(X)$ and $numbervar(Y)$ are respectively $VAR0$ and $VAR1$. On the other hand, in terms $f(X, a)$ and $f(Y, 1)$, $numbervar(X)$ and $numbervar(Y)$ are both $VAR0$. This is why the child node $VAR0$ of $f/2$ from Figure 3.4(c) is common to both terms.

3.3.2 Using Tries to Organize the Table Space

We next describe how tries are used to implement the table space. Figure 3.5 shows an example for a tabled predicate $f/2$ after the execution of the following tabling operations:

```
tabled_subgoal_call: f(X,a)
tabled_subgoal_call: f(Y,1)
tabled_new_answer:   f(0,a)
tabled_new_answer:   f(a,1)
tabled_new_answer:   f(b,1)
```

We use two levels of tries: one stores the subgoal calls, the other the answers. Each different call to a tabled predicate corresponds to a unique path through the *subgoal trie structure*. Such a path always starts from the root node in this trie, the `SG_TRIE`

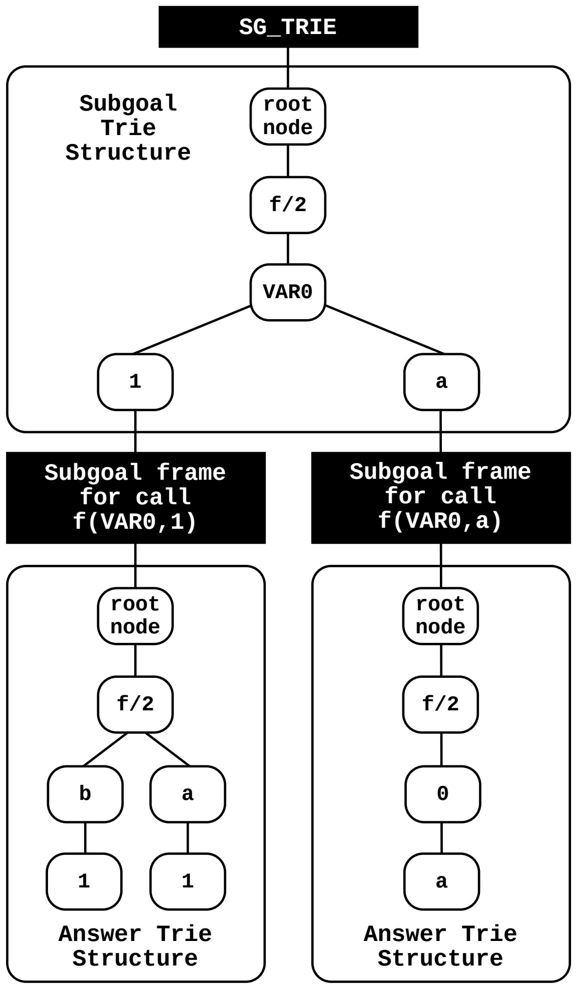


Figure 3.5: Using tries to organize the table space

variable, follows a sequence of the *subgoal trie nodes*, and terminates at a leaf data structure, the *subgoal frame*. Each subgoal frame stores information about the subgoal, namely an entry point to its *answer trie structure*. Each unique path through the *answer trie nodes* corresponds to a different answer to the entry subgoal.

3.3.3 Implementation Details

Tries are implemented by representing each trie node by a data structure with four fields each. The first field (`TrNode_symbol`) stores the symbol for the node. The second (`TrNode_child`) and third (`TrNode_parent`) fields store pointers respectively to the first child node and to the parent node. The fourth field (`TrNode_sibling`) stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and then the list of sibling pointers. Figure 3.6 illustrates the actual implementation for the trie presented in Fig. 3.4(c). Observe that we can collect all children of a node, and that we can always reach the root from a leaf node.

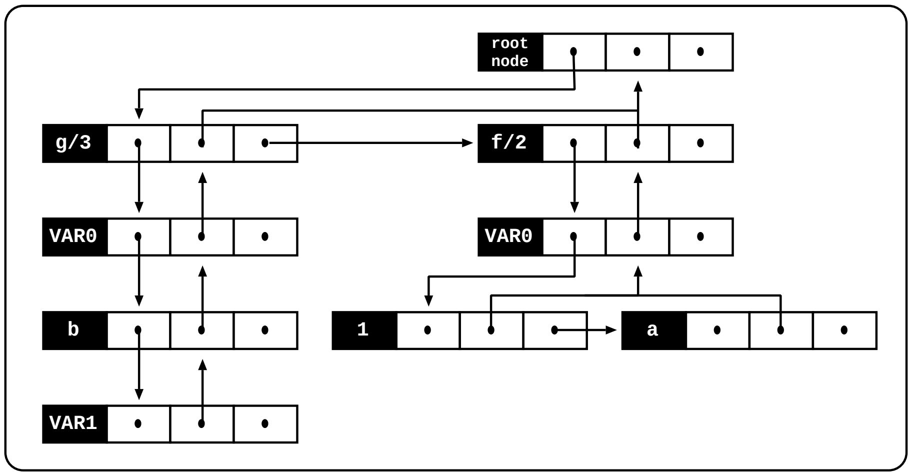


Figure 3.6: The implementation of the trie in Fig. 3.4(c)

Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_check_insert()` procedure for each symbol that represents the term being checked. Given a node `PARENT` and a symbol `S`, the procedure returns the child node of `PARENT` that represents the given symbol `S`. Figure 3.7 shows the pseudo-code. Initially it traverses the chain of sibling nodes that represent alternative paths from the given parent node and checks for one representing the given symbol. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is allocated and inserted in the beginning of the chain. To allocate new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `TrNode_symbol`, `TrNode_child`, `TrNode_parent` and `TrNode_sibling` fields of the new allocated node.

```
trie_check_insert(TrNode PARENT, YAP_Term S) {
  child = TrNode_child(PARENT);
  while (child) {              // check if a node for S was already inserted
    if (TrNode_symbol(child) == S)
      return child;                                         // node found
    child = TrNode_sibling(child);
  }
  child = new_trie_node(S, NULL, PARENT, TrNode_child(PARENT));
  TrNode_child(PARENT) = child;                 // insert the new node for S
  return child;
}
```

Figure 3.7: Pseudo-code for `trie_check_insert()`

Searching through a chain of sibling nodes that represent alternative paths is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (8 in our implementation) controls whether to dynamically index the nodes through a hash table, hence providing direct node access and optimising search. Further hash collisions are reduced by dynamically expanding the hash tables. For simplicity of presentation, Fig. 3.7 omits the hashing mechanism.

To manipulate tries we have defined three interface procedures. The `open_trie()` procedure initializes a new trie structure and returns the reference to the root node of the new trie. New terms are stored using the `put_trie_entry(TrNode root, YAP_Term term)` procedure, where `root` is the root node of the trie to be used and `term` is the term to be inserted. It returns the reference to the leaf node of the inserted term. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node. To load a term from a trie we have defined the `get_trie_entry(TrNode leaf)` procedure, where `leaf` is the reference to the leaf node of the term to be returned. When loading a term, the trie nodes for the term in hand are traversed in bottom-up order. The trie structure is not traversed in a top-down manner because the insertion and retrieval of terms is an asynchronous process, new trie nodes may be inserted at *anytime* and *anywhere* in the trie structure. This induces complex dependencies that limit the efficiency of alternative top-down loading schemes.

When inserting terms in the table space we need to distinguish to situations: **(i)** inserting tabled calls in the subgoal trie structure; and **(ii)** inserting answers in a particular answer trie structure. The first situation is handled by the `put_tabled_call()` procedure as shown in Fig. 3.8.

```
put_tabled_call(YAP_Term SUBGOAL_CALL) {
  leaf = put_trie_entry(SG_TRIE, SUBGOAL_CALL);
  sf = TrNode_child(leaf);
  if (sf == NULL) {                                      // new subgoal call
    sf = new_subgoal_frame();
    TrNode_child(leaf) = sf;
  }
  return sf;                   // sf is the subgoal frame for the tabled call
}
```

Figure 3.8: Pseudo-code for `put_tabled_call()`

The procedure starts by inserting the call in the subgoal trie structure (remember from Fig. 3.5 that `SG_TRIE` holds the root node for the subgoal trie structure) and then it checks if the `TrNode_child` field of the returned leaf node is `NULL`. When this is not the case, then the call is a repeated call and a subgoal frame is already stored. Otherwise, the `TrNode_child` field of the leaf node is made to point to a new subgoal frame data structure.

Inserting answers in an answer trie structure is handled by the `put_answer()` procedure as shown next in Fig. 3.9. The procedure starts by inserting the answer in the answer trie structure for the subgoal frame in hand (the `SgFr_answers` field holds the root node for the answer trie) and then it checks if the returned leaf node corresponds to a new answer. When new answers are inserted, the `TrNode_child` field of the leaf nodes is used to chain the answers in insertion time order, so that we can recover them in the same order they were inserted. The subgoal frame points to the first and last answer in this chain (the `SgFr_first_answer` and `SgFr_last_answer` fields). Thus, when a repeated call appears, a consumer node only needs to point at the leaf node for its last consumed answer, and consumes more answers just by following the chain. The `get_trie_entry()` procedure is then used to consume each answer.

```
put_answer(SgFr SUBGOAL_FRAME, YAP_Term ANSWER) {
  leaf = put_trie_entry(SgFr_answers(SUBGOAL_FRAME), ANSWER)
  if (TrNode_child(leaf) == NULL &&
      leaf != SgFr_last_answer(SUBGOAL_FRAME)) {               // new answer
    if (SgFr_first_answer(SUBGOAL_FRAME) == NULL) {          // first answer
      SgFr_first_answer(SUBGOAL_FRAME) = leaf;
    } else {
      TrNode_child(SgFr_last_answer(SUBGOAL_FRAME)) = leaf;
    }
    SgFr_last_answer(SUBGOAL_FRAME) = leaf;
    return TRUE;
  }
  return FALSE;                                            // repeated answer
}
```

Figure 3.9: Pseudo-code for `put_answer()`

Chapter 4

Tabled Evaluation with Continuation Calls

This chapter describes the design and implementation of the first tabling mechanism that we have implemented, we name it *tabled evaluation with continuation calls*. This mechanism is based on SLG resolution [CW96], as described in subsection 2.2.1, and follows a scheduling strategy based on the *local scheduling* strategy as introduced in subsection 2.2.3.

First, we briefly describe the basic execution model for this tabling mechanism. Next, we show how a tabled program is transformed to include specific tabling primitives that provide direct control over the search strategy and, we present an example that shows the interaction between Prolog execution and the tabling primitives for this mechanism. We then provide the details for implementing this mechanism as an external Prolog module written in C and discuss how completion is detected.

4.1 Basic Execution Model

The tabled evaluation with continuation calls is a *delaying-based* tabling mechanism: in order to ensure that all answers are correctly returned to all consumers and therefore detect completion, it needs to preserve the computation state of suspended tabled subgoal calls. The basic idea is as follow. Whenever a tabled subgoal is first called, a new entry is allocated in the table space and the evaluation begins with a generator node exploring the first clause for the corresponding tabled predicate. On the other hand, repeated calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. When no more unconsumed answers are available, the computation state for the repeated call is *suspended*. In this model, suspension is implemented by leaving a *continuation call* for the current computation in the table entry corresponding to the repeated call being suspended.

During this process and as further new answers are found, they are stored in their tables and returned to all repeated calls by calling the previously stored continuation calls. We then follow a local scheduling approach and execution *fails* for the new answer operation. New answers are only returned to the calling environment when all program clauses for the subgoal at hand are resolved.

Therefore, when the execution backtracks to a generator node, we first exhaust the remaining clauses and only then, when no more clauses are available, we start consuming the available answers for the subgoal. After consuming all answers, if the generator depends on older subgoals, we then proceed as for consumers and we fail by leaving the continuation call for the current computation in the table entry corresponding to the current generator node. It is only when a tabled subgoal is marked as complete that the corresponding continuation calls are deleted.

4.2 Program Transformation

To deal with tabled predicates and their answers, we use specific tabling primitives that provide direct control over the search strategy. A tabled logic program is first transformed to include the tabling primitives through source level transformations and only then, the resulting program is compiled.

Program transformation only applies to clauses of predicates previously declared as tabled predicates. Given a clause of a tabled predicate, each call to a tabled predicate in its body is replaced by a `tabled_call/5` primitive. This primitive implements the tabled subgoal call operation as described above. In addition, a `new_answer/2` primitive is added to the end of each clause body. The `new_answer/2` primitive is responsible to check for redundant answers and to return new answers to the stored continuation calls.

A major issue with this tabling mechanism is how to deal with continuation calls. Intuitively, the continuation call for a generator or consumer node is the code to be executed when an answer is returned to the node. With respect to a clause and an occurrence of a tabled predicate in the clause body, the continuation call is the portion of the clause body after the tabled predicate literal. The problem here is that later, when calling a continuation call, we cannot execute a clause starting from the literal after the corresponding tabled predicate. Our program transformation captures the continuation of a tabled predicate in a clause body by introducing a new predicate symbol, which has a single clause and whose body is the continuation to be executed when an answer is returned.

Figure 4.1 shows how the right recursive `path/2` program from Fig. 2.3 is transformed and augmented with tabling primitives to implement tabled evaluation with continuation calls.

```
path(X,Z):- tabled_call(path(X,Z),Sid,_,path0,true),
            consume_answer(path(X,Z),Sid).

path0(path(X,Z),Sid):-
            edge(X,Y),
            tabled_call(path(Y,Z),Sid,[X,Z,Y],path0,path1).
path1(path(Y,Z),Sid,[X,Z,Y]):-
            new_answer(path(X,Z),Sid).
path0(path(X,Z),Sid):-
            edge(X,Z),
            new_answer(path(X,Z),Sid).
```

Figure 4.1: Program transformation for the right recursive `path/2` program

A `path/2` clause is maintained so that tabled predicate `path/2` can be called from other predicates without any change. The tabling primitive `tabled_call/5` in the body of `path/2` starts the tabled evaluation process and ensures that the subgoal will be completely evaluated. The `consume_answer/2` primitive then returns each computed answer one at a time.

Each clause in the original definition of `path/2` becomes a clause for a new distinct predicate, `path0/2` in the example, with 2 arguments. The first argument is the previous head clause; the second is the *subgoal id*. The id for a subgoal call is generated by the `tabled_call/5` primitive and is used to guarantee that the answers found for a call are properly saved within that call. When the `tabled_call/5` primitive is called for a new subgoal (the first argument), a new entry is allocated in the table space and an unique id is returned (`Sid`). This id is then used by the `tabled_call/5` primitive to start the evaluation process by calling the predicate of arity 2 represented in the forth argument, `path0` in this case, with the appropriate arguments.

Each continuation of a tabled predicate in a clause body also becomes a clause for a new distinct predicate. In the example of Fig. 4.1, we only have one such predicate, `path1/3`. The first two arguments of `path1/3` are the same as the arguments of `path0/2`. The third argument is a list of variables used to pass variable bindings across continuation calls.

Continuations calls are added by the `tabled_call/5` primitive when a computation is being suspended. A continuation call is a triplet formed by the second, third and fifth arguments passed to the `tabled_call/5`. The second argument stores the id of the call that is calling the `tabled_call/5` in the body of a clause. The third argument stores the bindings for the variables appearing in the head and in the body of the clause. The fifth argument stores the predicate of arity 3 to be called in the continuation. When a new answer is found, each continuation triplet is then used in conjunction with the new answer to construct the calls that continue the suspended computations.

Next we present two more examples that better illustrate how a program is transformed to include specific tabling primitives for the tabled evaluation with continuation calls mechanism. Figure 4.3 shows the transformed program for the left recursive `path/2` definition in Fig. 4.2, and Fig. 4.5 shows the transformed program for the doubly recursive `path/2` definition in Fig. 4.4.

```
:- table path/2

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

Figure 4.2: The left recursive `path/2` program

```
path(X,Z):- tabled_call(path(X,Z),Sid,_,path0,true),
            consume_answer(path(X,Z),Sid).

path0(path(X,Z),Sid):-
            tabled_call(path(X,Y),Sid,[X,Z,Y],path0,path1).
path1(path(X,Y),Sid,[X,Z,Y]):-
            edge(Y,Z),
            new_answer(path(X,Z),Sid).
path0(path(X,Z),Sid):-
            edge(X,Z),
            new_answer(path(X,Z),Sid).
```

Figure 4.3: Program transformation for the left recursive `path/2` program

```
:- table path/2

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

Figure 4.4: The doubly recursive `path/2` program

```
path(X,Z):- tabled_call(path(X,Z),Sid,_,path0,true),
            consume_answer(path(X,Z),Sid).

path0(path(X,Z),Sid):-
            tabled_call(path(X,Y),Sid,[X,Z,Y],path0,path1).
path1(path(X,Y),Sid,[X,Z,Y]) :-
            tabled_call(path(Y,Z),Sid,[X,Z,Y],path0,path2).
path2(path(Y,Z),Sid,[X,Z,Y]):-
            new_answer(path(X,Z),Sid).
path0(path(X,Z),Sid):-
            edge(X,Z),
            new_answer(path(X,Z),Sid).
```

Figure 4.5: Program transformation for the doubly recursive `path/2` program

4.3 An Evaluation Example

Consider again the program transformation in Fig. 4.1 for the right recursive `path/2` definition of Fig. 2.3. Figure 4.6 shows the evaluation sequence for the query goal `p(1,Z)` if applying tabled evaluation with continuation calls.

At the top, the figure illustrates the program code and the final state of the table space at the end of the evaluation. The bottom sub-figure shows the resulting forest of trees with the numbering of nodes denoting the evaluation sequence. For illustration purposes the program code was simplified, predicates `path/2`, `path0/2`, `path1/3` and `edge/2` are denoted as `p/2`, `p0/2`, `p1/3` and `e/2`, respectively.

Let us examine the evaluation in more detail. The evaluation begins by calling the `tabled_call/5` primitive for the `p(1,Z)` subgoal. The `p(1,Z)` subgoal is a new tabled subgoal call and thus, a new entry is allocated in the table space for it, with id `sid1`, and a new generator node is created (node 1). Generator nodes are represented by black oval boxes. Next, `p0(p(1,Z),sid1)` is called, creating node 2. The execution then proceeds with the first alternative of `p0/2`, calling `e(1,Y)` that binds `Y` with `2` and with primitive `tabled_call/5` being called for the `p(2,Z)` subgoal (step 4). As this is the first call to `p(2,Z)`, we add a new entry for it, with id `sid2`, and proceed by allocating a new generator node as shown in the bottommost tree.

```
p(X,Z):- tabled_call(p(X,Z),Sid,_,p0,true), consume_answer(p(X,Z),Sid).

p0(p(X,Z),Sid):- e(X,Y), tabled_call(p(Y,Z),Sid,[X,Z,Y],p0,p1).
p1(p(Y,Z),Sid,[X,Z,Y]):- new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid):- e(X,Z), new_answer(p(X,Z),Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Continuation Calls
sid1	1. p(1,Z)	15. p(1,1) 23. p(1,2) 32. complete	9. p1(?ANS?,sid2,[2,Z,1])
sid2	4. p(2,Z)	12. p(2,1) 25. p(2,2) 32. complete	20. p1(?ANS?,sid1,[1,Z,2])

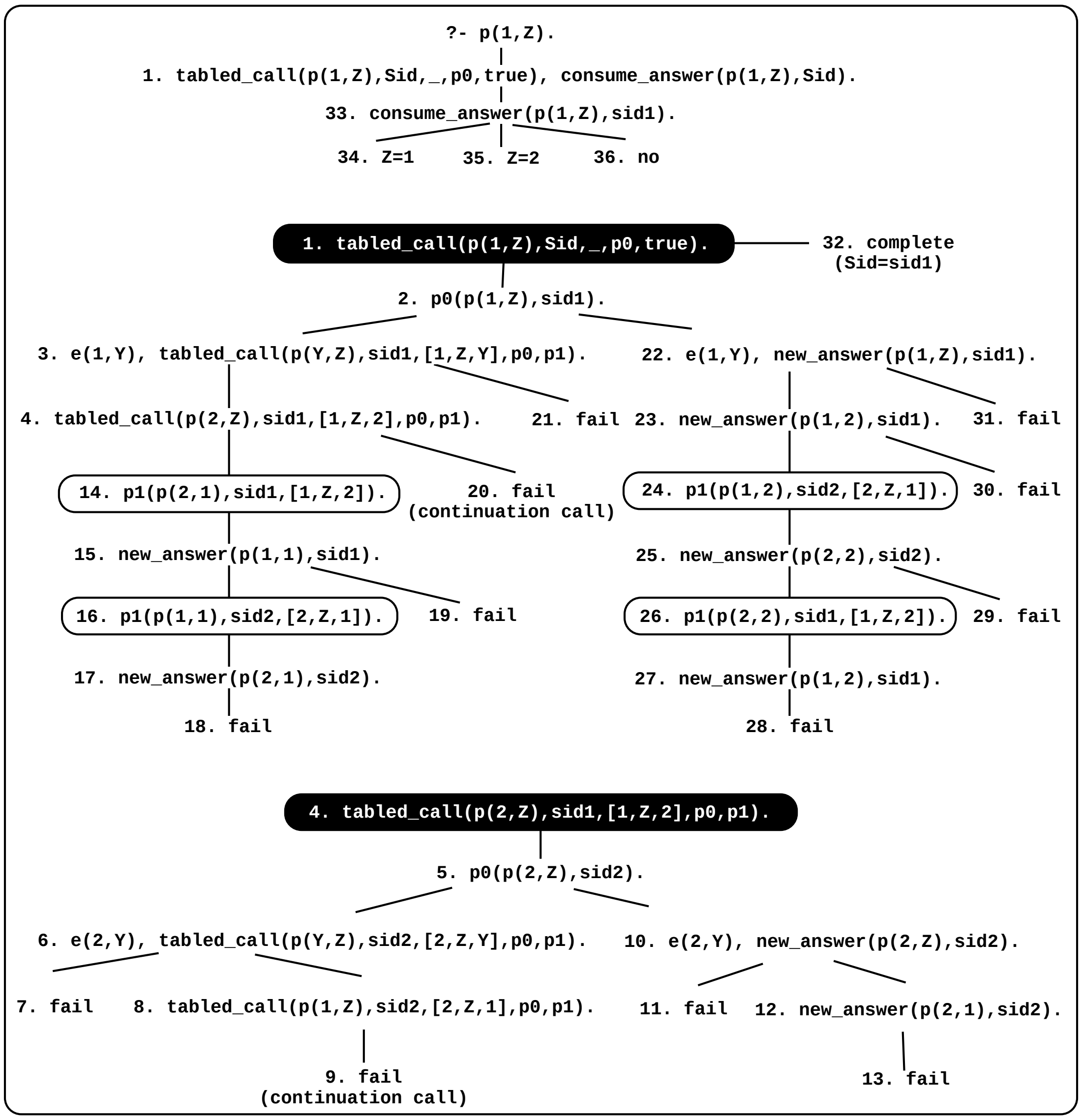

?- p(1,Z).
1. tabled_call(p(1,Z),Sid,_,p0,true), consume_answer(p(1,Z),Sid).
33. consume_answer(p(1,Z),sid1).
34. Z=1
35. Z=2
36. no
1. tabled_call(p(1,Z),Sid,_,p0,true).
32. complete (Sid=sid1)
2. p0(p(1,Z),sid1).
3. e(1,Y), tabled_call(p(Y,Z),sid1,[1,Z,Y],p0,p1).
22. e(1,Y), new_answer(p(1,Z),sid1).
4. tabled_call(p(2,Z),sid1,[1,Z,2],p0,p1).
21. fail
23. new_answer(p(1,2),sid1).
31. fail
14. p1(p(2,1),sid1,[1,Z,2]).
20. fail (continuation call)
24. p1(p(1,2),sid2,[2,Z,1]).
30. fail
15. new_answer(p(1,1),sid1).
25. new_answer(p(2,2),sid2).
16. p1(p(1,1),sid2,[2,Z,1]).
19. fail
26. p1(p(2,2),sid1,[1,Z,2]).
29. fail
17. new_answer(p(2,1),sid2).
27. new_answer(p(1,2),sid1).
18. fail
28. fail
4. tabled_call(p(2,Z),sid1,[1,Z,2],p0,p1).
5. p0(p(2,Z),sid2).
6. e(2,Y), tabled_call(p(Y,Z),sid2,[2,Z,Y],p0,p1).
10. e(2,Y), new_answer(p(2,Z),sid2).
7. fail
8. tabled_call(p(1,Z),sid2,[2,Z,1],p0,p1).
11. fail
12. new_answer(p(2,1),sid2).
9. fail (continuation call)
13. fail

Figure 4.6: Tabled evaluation with continuation calls for the right recursive `path/2` program

Again, `p(2,Z)` is resolved against the first clause for `p0/2` (step 5). Then, the first clause for `e(2,Y)` fails (step 7), but the second succeeds by binding `Y` with `1` (step 8). The `tabled_call/5` primitive is then called again for `p(1,Z)`. Since `p(1,Z)` is a repeated subgoal call and no answers are still available for it, the current evaluation is *suspended*. A triplet formed by the id `sid2`, the list `[2,Z,1]`, and by the predicate symbol `p1` is then stored in the table entry for `p(1,Z)` as a continuation call (step 9).

We then return to node 5 and try the second clause for `p0/2`, obtaining a first answer for `p(2,Z)` (step 12). The answer is inserted in the table and, because there are no continuation calls for `p(2,Z)`, the execution fails. Remember that the new answer operation always fails when using a local scheduling approach. The execution then backtracks to node 4 and we check whether `p(2,Z)` can be completed. It can not, because it depends on the continuation call left by subgoal `p(1,Z)` at step 9. If that continuation call gets executed, further answers for `p(2,Z)` can be found.

At that point, the answers in the table entry for `p(2,Z)` should be consumed. Remember that answers are only returned to a generator node when all program clauses for it were resolved. Consuming an answer corresponds to the execution of the continuation call for the `tabled_call/5` at hand. The continuation call for the `tabled_call/5` primitive at node 4 is thus executed for the answer `p(2,1)` (step 14). Continuation calls are represented by white oval boxes.

A first answer, `p(1,1)`, is then found for `p(1,Z)` (step 15) and, because there is a continuation call for `p(1,Z)`, the execution continues by calling it with the newly found answer (step 16). A redundant answer is then found for `p(2,Z)` (step 17), so we fail and backtrack again to node 4. We then proceed as for consumers and we fail by leaving the continuation call for the current computation in the table entry for `p(2,Z)` (step 20). The evaluation then explores the second clause at node 2, obtaining a second answer for `p(1,Z)` (step 23) and a second answer for `p(2,Z)` (step 25). When backtracking to node 1, the subgoals `p(1,Z)` and `p(2,Z)` are now fully exploited. So, we declare the two subgoals to be completed (step 32).

Again, at that point, the answers in the table entry should be consumed. However, the continuation call in the `tabled_call/5` primitive at node 1 is the predicate symbol `true`. This means that the `tabled_call/5` primitive was executed from the clause representing the original `p/2` predicate. Therefore, instead of consuming answers, we simply succeed by binding `Sid` with the subgoal id for `p(1,Z)`, `sid1` in this case. The `consume_answer/2` primitive then returns by backtracking the computed answers for the `sid1` tabled subgoal call.

4.4 Implementation Details

Next we describe in more detail all the implementation issues required to fully suport tabled evaluation with continuation calls.

4.4.1 Subgoal Frames

A key data structure in the table space organization is the *subgoal frame*. Remember from subsection 3.3.2 that subgoal frames are used to store information about the tabled subgoals and to act like entry points to the trie structures where answers are stored. Whenever the `put_tabled_call()` is executed for a new tabled subgoal (please see Fig. 3.8 for more details), the `new_subgoal_frame()` procedure is called in order to allocate and initialize a new subgoal frame. Figure 4.7 shows its implementation for the current tabling mechanism.

```
new_subgoal_frame() {
  sf = allocate_subgoal_frame();
  SgFr_state(sf) = READY;
  SgFr_answers(sf) = open_trie();
  SgFr_first_answer(sf) = NULL;
  SgFr_last_answer(sf) = NULL;
  SgFr_cont_calls(sf) = NULL;
  SgFr_previous(sf) = SF_TOP;
  if (SF_TOP == NULL)
    SgFr_dfn(sf) = 1;
  else
    SgFr_dfn(sf) = SgFr_dfn(SF_TOP) + 1;
  SgFr_dep(sf) = SgFr_dfn(sf);
  SF_TOP = sf;
  return sf;
}
```

Figure 4.7: Pseudo-code for `new_subgoal_frame()`

A subgoal frame is a eight field data structure. These fields have the following meaning:

- `SgFr_state:` indicates the state of the subgoal. During evaluation, a subgoal can be in one of the following states: *ready*, *evaluating* or *complete*.
- `SgFr_answers:` is the entry point to the answer trie structure.
- `SgFr_first_answer:` is the pointer to the first inserted answer in the answer trie or `NULL` if no answers are available.

- `SgFr_last_answer:` is the pointer to the last inserted answer in the answer trie or `NULL` if no answers are available.

- `SgFr_cont_calls:` is the pointer to continuation calls associated with the subgoal or `NULL` if no continuation calls are available.

- `SgFr_previous:` is the pointer to the previously allocated subgoal frame that is still not completed. A global variable, `SF_TOP`, always points to the youngest subgoal frame being evaluated.

- `SgFr_dfn:` is the *depth-first number* of the call. New subgoal frames are numbered incrementally and according to the order in which they appear in the evaluation.

- `SgFr_dep:` is the *depth-first number* of the older call in which the current call depends. It is initialized with the same number as its depth-first number, meaning that no dependencies exist. It is critical to the fix-point check procedure that we discuss later.

Figure 4.8 uses the example from Fig. 4.6 to illustrate how these fields are used and updated during a tabled evaluation.

Figure 4.8(a) shows the state of the table space after calling the `tabled_call/5` primitive for the `p(1,Z)` subgoal (step 1 in Fig. 4.6). The `p(1,Z)` subgoal is a new tabled subgoal and thus, a new subgoal frame is allocated and initialized in the table space for it. Next, `tabled_call/5` is called for `p(2,Z)` and a new subgoal frame is also allocated and initialized (step 4 in Fig. 4.6). The `SF_TOP` variable is made to point to this new frame. The `tabled_call/5` primitive is then called again for `p(1,Z)` and the evaluation is suspended (step 9 in Fig. 4.6). The `SgFr_dep` field of the subgoal frame for `p(2,Z)` is updated to 1, the depth-first number of `p(1,Z)`, and the continuation call for the current computation is stored in the subgoal frame for `p(1,Z)`. Figure 4.8(b) shows the resulting state of both subgoal frames at that point.

The execution then continues and first answers for `p(2,Z)` and `p(1,Z)` are found and inserted in the answer tries starting from the corresponding subgoal frames (steps 12 and 15 in Fig. 4.6). The `SgFr_first_answer` and `SgFr_last_answer` fields of each frame are both made to point to the leaf node of the corresponding first answer. In the continuation, we fail by leaving the continuation call for the current computation in the subgoal frame for `p(2,Z)` (step 20 in Fig. 4.6). Figure 4.8(c) shows the resulting state of the subgoal frame for `p(2,Z)` at that point.

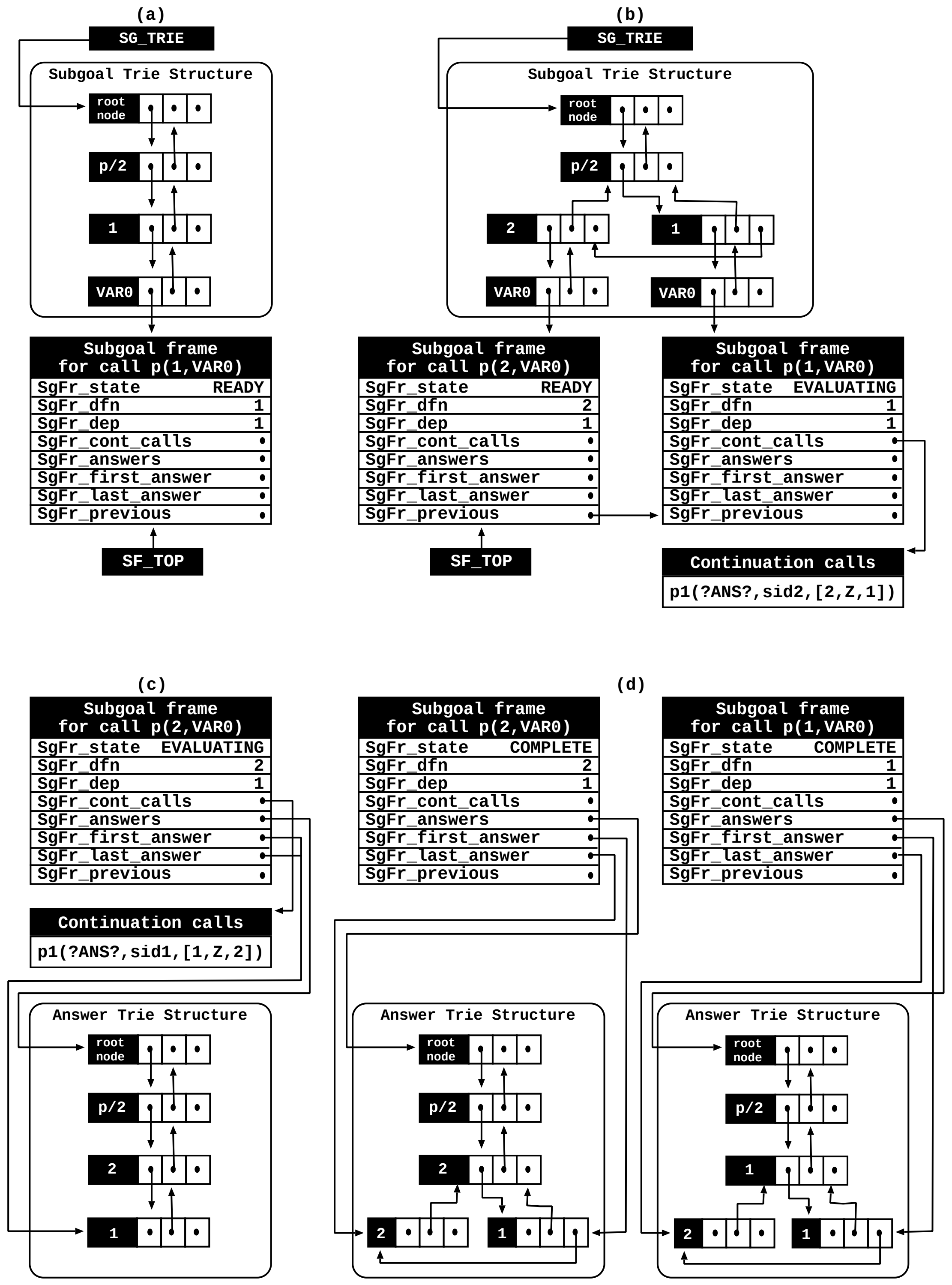


Figure 4.8: Subgoal frames at different points of the evaluation of the right recursive `path/2` program of Fig. 4.6

The evaluation then obtains second answers for `p(1,Z)` and `p(2,Z)` (steps 23 and 25 in Fig. 4.6) and, at the end, both subgoals are declared to be completed (step 32 in Fig. 4.6). This includes marking the corresponding subgoal frames as complete in their `SgFr_state` fields, deleting the pending continuation calls, and updating the `SF_TOP` variable to `NULL` as no more subgoals are being evaluated. Figure 4.8(d) shows the final state of both subgoal frames.

4.4.2 Tabling Primitives

Next we show the implementation details for the tabling primitives that support the tabled evaluation with continuation calls mechanism. We start with Fig. 4.9 showing the pseudo-code for the `tabled_call/5` primitive.

The `tabled_call/5` primitive starts by calling the `put_tabled_call()` procedure to insert the given `SUBGOAL_CALL` in the subgoal trie structure. Then, it checks if the resulting subgoal frame is new, that is, if the `SgFr_state` is `READY`. Being this the case, it changes the subgoal's state to `EVALUATING`, constructs the call that starts the evaluation process and calls the Prolog engine to execute it. When returning, it checks for completion. If the `SgFr_dfn` and `SgFr_dep` fields are equal then we know that we are in a leader node position because no dependencies exist and, therefore, all younger subgoals can be completed.

Otherwise, if the subgoal is not new or if the `SgFr_dfn` and `SgFr_dep` fields are different, we propagate the dependency in the `SgFr_dep` field of the current subgoal frame to the `SgFr_dep` field of the `CONT_SF` subgoal frame that continues the execution (please see subsection 4.4.3 for more details).

In both situations, we then consume the available answers, leave a continuation call if the subgoal is not completed, and fail. The only situation where this primitive does not fails is when it is executed from the clause representing the original tabled predicate, that is, where the `CONT_PRED` is `true`. In such situations, the `CONT_SF` argument is unified with the current subgoal frame pointer and the procedure succeeds. The `consume_answer/2` primitive then uses the given pointer to return the computed answers to the environment of the clause representing the original tabled predicate. Figure 4.10 shows the pseudo-code for the `consume_answer/2` primitive.

The `consume_answer/2` primitive was implemented in Yap as a backtrackable predicate. The `consume_answer_init()` procedure is executed when the primitive is first called, and the `consume_answer_cont()` procedure is executed when backtrack-

```
tabled_call(YAP_Term SUBGOAL_CALL, SgFr CONT_SF, YAP_Term CONT_VARS,
            YAP_Atom INITIAL_PRED, YAP_Atom CONT_PRED) {
  sf = put_tabled_call(SUBGOAL_CALL);

  if (SgFr_state(sf) == READY) {                           // new subgoal call
    SgFr_state(sf) = EVALUATING;
    initial_call = construct_call(INITIAL_PRED, SUBGOAL_CALL, sf);
    YAP_CallProlog(initial_call);
    if (SgFr_dfn(sf) == SgFr_dep(sf)) {                // check for completion
      do {
        SgFr_state(SF_TOP) = COMPLETE;
        delete_continuation_calls(SF_TOP);
        SF_TOP = SgFr_previous(SF_TOP);
      } while (SF_TOP != SgFr_previous(sf))  // complete all frames up to sf
    }
  }

  if (SgFr_state(sf) != COMPLETE)       // propagate dependencies to CONT_SF
    SgFr_dep(CONT_SF) = minimum(SgFr_dep(CONT_SF), SgFr_dep(sf));
  else if (SgFr_state(sf) == COMPLETE && CONT_PRED == true) {     // succeed
    YAP_Unify(CONT_SF, sf);
    return TRUE;
  }

  leaf = SgFr_first_answer(sf);                // get first answer leaf node
  while (leaf != NULL) {
    answer = get_trie_entry(leaf);              // load answer from the trie
    cont_call = construct_call(CONT_PRED, answer, CONT_SF, CONT_VARS);
    YAP_CallProlog(cont_call);
    leaf = TrNode_child(leaf);                  // get next answer leaf node
  }

  if (SgFr_state(sf) != COMPLETE)               // leave a continuation call
    add_continuation_call(sf, CONT_PRED, CONT_SF, CONT_VARS);

  return FALSE;                                    // always fail at the end
}
```

Figure 4.9: Pseudo-code for the tabled_call/5 primitive

ing occurs. To obtain by backtracking the full set of answers, we need to preserve the last returned answer. This is done by using the YAP_PRESERVE_DATA() and YAP_PRESERVED_DATA() macros as previously described in subsection 3.2.2.

We end with Fig. 4.11 that shows the pseudo-code for the new_answer/2 primitive.

The new_answer/2 primitive starts by calling the put_answer() procedure to insert the given ANSWER in the answer trie structure for the SF subgoal frame. If the answer is redundant, we simply fail. Otherwise, the answer is new and each continuation triplet stored in SF is then used in conjunction with the new answer to construct the calls

```
consume_answer_init(YAP_Term SUBGOAL_CALL, SgFr SF) {
  leaf = SgFr_first_answer(SF);                // get first answer leaf node
  if (leaf == NULL) {                                          // no answers
    YAP_cut_fail();
    return FALSE;
  }
  YAP_PRESERVE_DATA(mem_space, TrNode);
  *mem_space = leaf;      // preserve the current leaf node for backtracking
  answer = get_trie_entry(leaf);                 // load answer from the trie
  YAP_Unify(SUBGOAL_CALL, answer);
  return TRUE;
}

consume_answer_cont(YAP_Term SUBGOAL_CALL, SgFr SF) {
  YAP_PRESERVED_DATA(mem_space, TrNode);
  leaf = *mem_space;
  leaf = TrNode_child(leaf);                     // get next answer leaf node
  if (leaf == NULL) {                                     // no more answers
    YAP_cut_fail();
    return FALSE;
  }
  *mem_space = leaf;        // update the current leaf node for backtracking
  answer = get_trie_entry(leaf);                 // load answer from the trie
  YAP_Unify(SUBGOAL_CALL, answer);
  return TRUE;
}
```

Figure 4.10: Pseudo-code for the `consume_answer/2` primitive

```
new_answer(YAP_Term ANSWER, SgFr SF) {
  if (put_answer(SF, ANSWER) == TRUE) {
    for each (cont_pred, cont_sf, cont_vars) in SgFr_cont_calls(SF) do {
      cont_call = construct_call(cont_pred, ANSWER, cont_sf, cont_vars);
      YAP_CallProlog(cont_call);
    }
  }
  return FALSE;                                   // always fail at the end
}
```

Figure 4.11: Pseudo-code for the `new_answer/2` primitive

that continue the suspended computations.

4.4.3 Detecting Completion

We check for completion when the computation returns to the `tabled_call/5` primitive after exhausting all alternative for the subgoal call at hand. We use the `SgFr_dfn` and the `SgFr_dep` fields of the subgoal frames to quickly determine whether a subgoal is a leader node. The SgFr_dfn field marks the order in which the subgoals in evaluation were called. New subgoal frames are numbered incrementally and the global variable SF_TOP always points to the youngest subgoal frame in evaluation. The SgFr_dep field holds the number of the older call in which it depends. It is initialized with the same number as the SgFr_dfn field, meaning that initially no dependencies exist.

When checking for completion and using this information from the subgoal frames, a subgoal can quickly determine whether it is a leader node. If the SgFr_dfn and SgFr_dep fields are equal then we know that during its evaluation no dependencies to older subgoals have appeared and thus the *Strongly Connected Component (SCC)* including the subgoals starting from the frame referred by SF_TOP up to the current subgoal can be completed. On the other hand, if the SgFr_dep field holds a number less than its depth-first number, then we cannot perform completion. Instead, we must propagate the current dependency to the subgoal call that continues the evaluation and only when the computation reaches the subgoal that does not depends on older subgoals we must perform completion.

Figures 4.12, 4.13 and 4.14 show three different examples that illustrate how the SgFr_dfn and SgFr_dep fields are used to detect completion. At the top, each figure shows the subgoal dependencies and the leader nodes (nodes filled with a black background). The black dots in the sub-figures below indicate the fields being updated at each step of the example.

Figure 4.12 represents again the evaluation example from Fig. 4.6. Initially, p(1,Z) is first called and a new subgoal frame is allocated with SgFr_dfn and SgFr_dep initialized with 1. Next, p(2,Z) is also called and a second frame is allocated with SgFr_dfn and SgFr_dep initialized with 2 (step 2 in Fig. 4.12). In the continuation, p(2,Z) calls p(1,Z) again which creates a dependency between subgoals p(2,Z) and p(1,Z). The SgFr_dep field of subgoal p(2,Z) is thus updated to represent this dependency to p(1,Z) (step 3 in Fig. 4.12). Then, when the computation returns to p(2,Z), we cannot perform completion because the SgFr_dfn and SgFr_dep fields of p(2,Z) are different. We thus propagate the dependency in the SgFr_dep field of p(2,Z) to p(1,Z) (step 4 in Fig. 4.12). This has no effect because the SgFr_dep field of p(1,Z) is already 1. At the end, when the computation returns to p(1,Z), the two subgoals are marked as complete and the SF_TOP global variable is updated (step 5 in Fig. 4.12).

Figure 4.13 shows an example for a graph with more edges as represented by the e/2 facts in the sub-figure at the top. As in the previous example, we start with p(1,Z) and p(2,Z) allocating two subgoal frames with SgFr_dfn and SgFr_dep initialized with

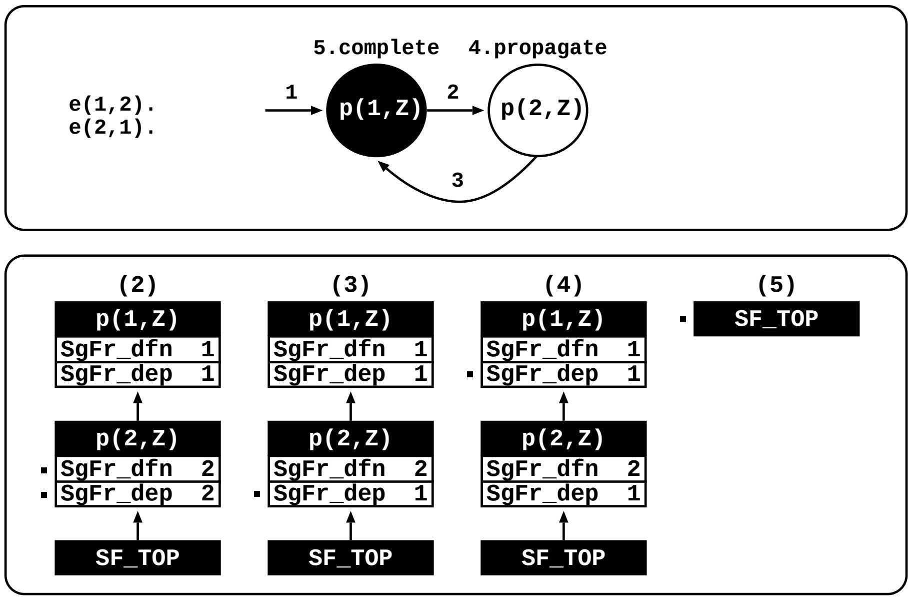


Figure 4.12: Detecting completion when evaluating the right recursive `path/2` program of Fig. 4.6

1 and 2, respectively. Next, `p(3,Z)` and `p(4,Z)` are also called and two new frames are allocated with `SgFr_dfn` and `SgFr_dep` initialized with 3 and 4, respectively. In the continuation, `p(4,Z)` calls `p(2,Z)` again, and the `SgFr_dep` field of subgoal `p(4,Z)` is updated to represent this dependency (step 5 in Fig. 4.13). Then, the computation returns to `p(4,Z)` and we propagate the dependency in the `SgFr_dep` field of `p(4,Z)` to `p(3,Z)` (step 6 in Fig. 4.13). The same happens when the computation returns to `p(3,Z)` and we propagate its dependency to `p(2,Z)` (step 7 in Fig. 4.13).

The interesting case occurs when the computation returns to `p(2,Z)`. The `SgFr_dfn` and `SgFr_dep` fields of `p(2,Z)` are equal which means that subgoal `p(2,Z)` is the leader node of the SCC that includes the subgoals starting from `SF_TOP` up to it, that is, subgoals `p(4,Z)`, `p(3,Z)` and `p(2,Z)`. The three subgoals are thus marked as complete and the `SF_TOP` variable is updated to the previous subgoal in evaluation, `p(1,Z)` in this case (step 8 in Fig. 4.13). At the end, the computation returns to `p(1,Z)` and its subgoal frame is marked as completed as no more dependencies exist. What this examples shows is that subgoals are not necessarily completed at the end of the evaluation, and that we can also have sets of subgoals being completed while others are still being evaluated.

Our last example shows a more complex graph as represented by the `e/2` facts in the sub-figure at the top of Fig. 4.14. Node 3 has two edges starting from it, one to node

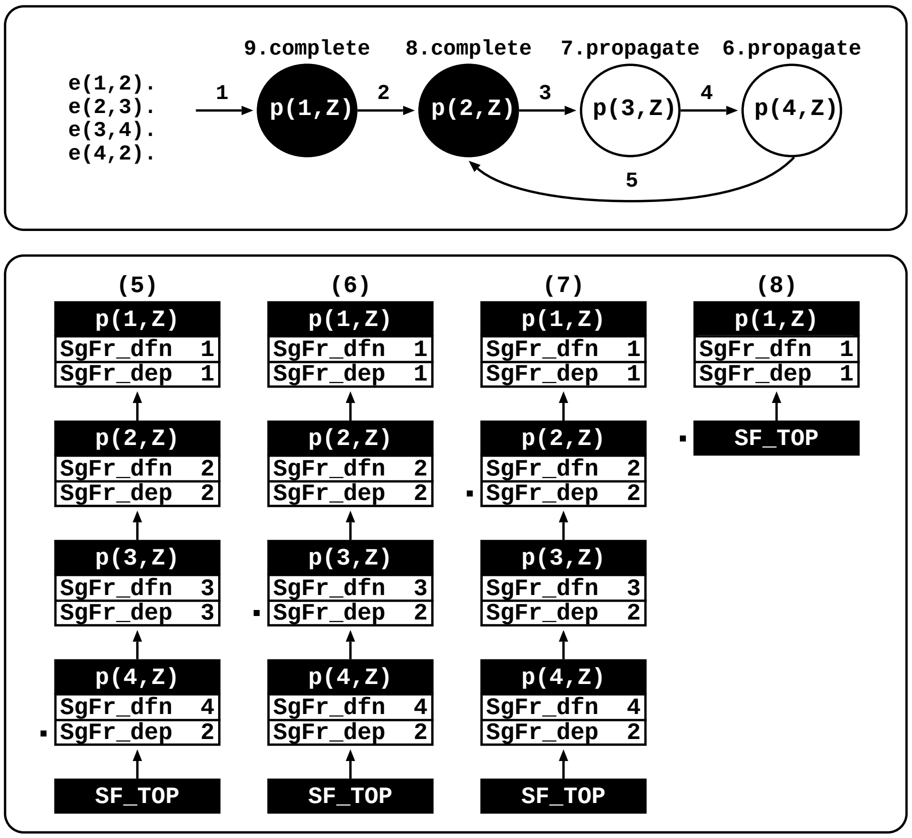


Figure 4.13: Detecting completion in the middle of the evaluation

4, as in the previous example, and an extra one to node 5. There is also a new edge from node 5 to node 1. As we will see this induces more than a single dependency during the evaluation.

Again, we start by allocating and initializing frames to subgoals `p(1,Z)`, `p(2,Z)`, `p(3,Z)` and `p(4,Z)`. Next, `p(4,Z)` calls `p(2,Z)` and the `SgFr_dep` field of subgoal `p(4,Z)` is updated to represent this dependency. Then, the computation returns to `p(4,Z)` and we propagate the dependency in the `SgFr_dep` field of `p(4,Z)` to `p(3,Z)` (step 6 in Fig. 4.14).

The interesting case occurs when the computation backtracks from `p(4,Z)`. The `e(3,Z)` call has two alternatives and thus, when backtracking, we take the second alternative, calling then `p(5,Z)`. A new subgoal frame is then allocated with `SgFr_dfn` and `SgFr_dep` initialized with 5 (step 7 in Fig. 4.14). Next, `p(5,Z)` calls `p(1,Z)` again, and the `SgFr_dep` field of subgoal `p(5,Z)` is updated to 1 (step 8 in Fig. 4.14). Then, the computation returns to `p(5,Z)` and we propagate the dependency in the `SgFr_dep` field of `p(5,Z)` to the continuation subgoal for `p(5,Z)`, that is, the subgoal passed in

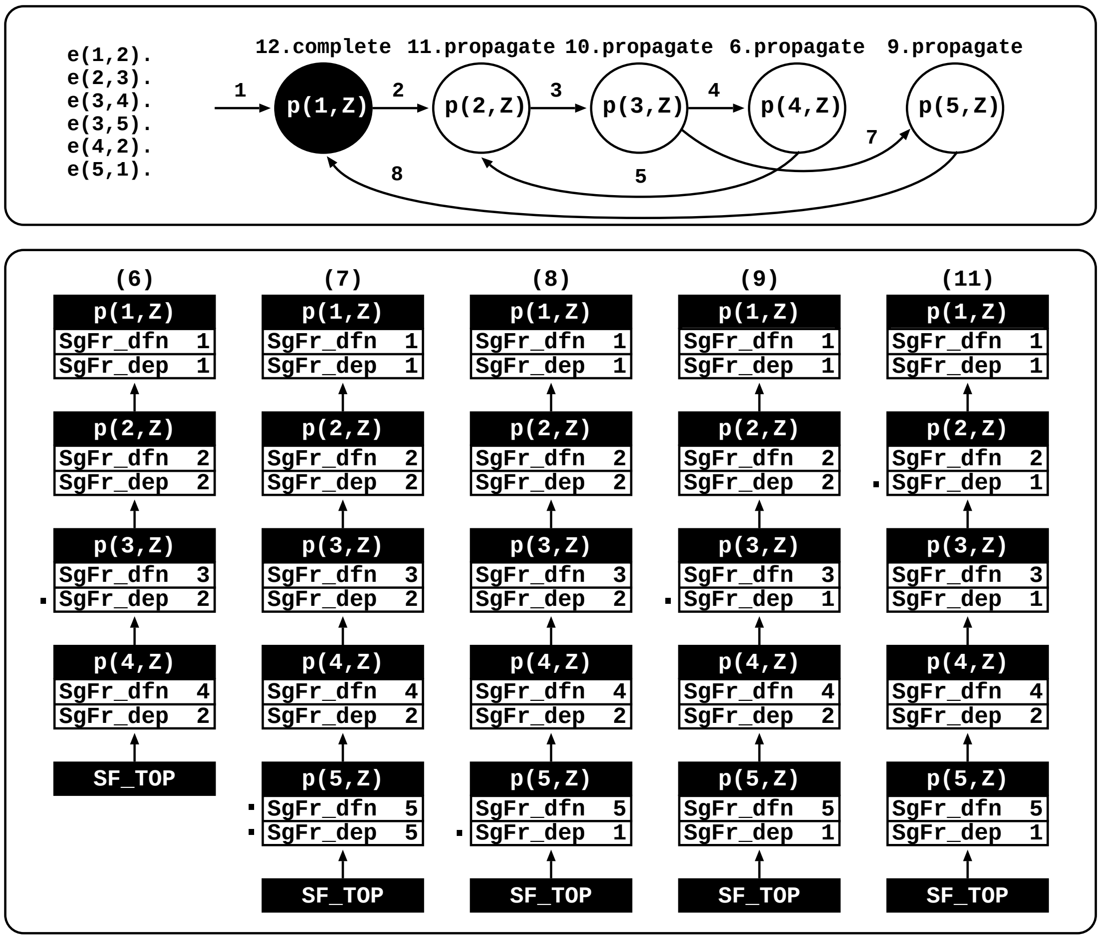


Figure 4.14: Detecting completion when propagating more than one dependency

the second argument of the `tabled_call/5` for `p(5,Z)`, `p(3,Z)` in this case (step 9 in Fig. 4.14). The subgoal frame for `p(3,Z)` already holds a dependency in its `SgFr_dep` field, but as the new dependency from `p(5,Z)` is older (the depth-first number is smaller) it is updated. This dependency is then propagated to `p(2,Z)` and next to `p(1,Z)` (step 11 in Fig. 4.14). At the end, the computation returns to `p(1,Z)` and all subgoals are marked as complete.

Chapter 5

Dynamic Reordering of Alternatives

This chapter describes the design and implementation of the second tabling mechanism that we have implemented, the DRA (Dynamic Reordering of Alternatives) linear tabling mechanism as proposed by Guo *et al.* [GG01]. As for the previous mechanism, we follow a scheduling strategy based on the local scheduling strategy.

Initially, we describe the basic execution model for the DRA technique and then we show how a tabled program is transformed to include specific tabling primitives for this mechanism. Next, we present an example showing the interaction between Prolog execution and the tabling primitives for DRA evaluation. We then provide the details for implementing this mechanism as an external Prolog module in Yap.

5.1 Basic Execution Model

The DRA technique is based on the *dynamic reordering of alternatives with repeated calls* for incorporating tabling into an existing logic programming system. The DRA technique not only memorizes the answers for the tabled subgoal calls, but also the alternatives leading to repeated calls, the *looping alternatives*. It then uses the looping alternatives to repeatedly recompute them until a fix-point is reached. During evaluation, a tabled call can be in one of three possible states: *normal state*, *looping state* or *complete state*. The state transition graph for DRA evaluation is shown next in Fig. 5.1

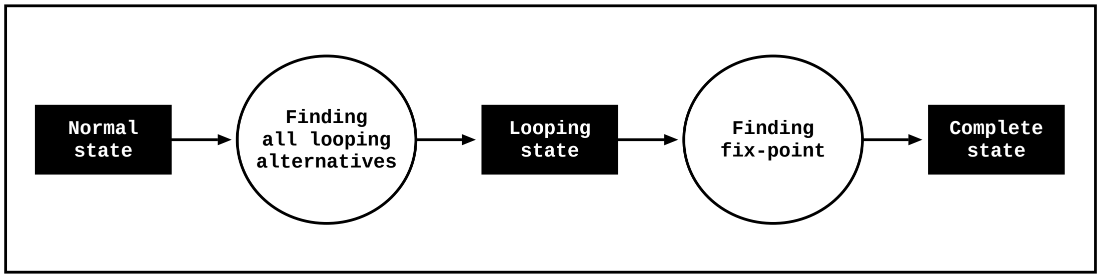


Figure 5.1: State transition graph

Consider a tabled subgoal call *C*. Initially, *C* enters in normal state where it is allowed to explore the matched clauses as in standard Prolog. In this state, while exploring the matching clauses, the model checks for looping alternatives. If a repeated call is found, the current clause that matches the original call to *C* will be memorized as a looping alternative. Essentially, the alternative corresponding to this call will be reordered and placed at the end of the alternative list for the call. This call will not be expanded at the moment because it can potentially lead to an infinite loop. Instead, it will consume the available answers for the call.

After exploring all the matched clauses, *C* goes into the looping state. From this point, it keeps trying the looping alternatives repeatedly until reaching a fix-point. If no new answers are found during one cycle of trying the looping alternatives, then we have reached a fix-point and we can say that *C* is completely evaluated.

The DRA scheme repeatedly tries looping alternatives. Retrying these alternatives may cause redundant computations: **(i)** a non-tabled subgoal in a looping alternative will be recomputed every time the looping alternative is tried; **(ii)** a repeated call to a tabled subgoal in a looping alternative will re-consume all tabled answers every time it is called. In fact, the problem of re-computation cannot be avoided and in the case of multiple dependent calls, the same computation has to be explored several times. One advantage is that only the looping alternatives are recomputed.

5.2 Program Transformation

Figure 5.2 uses the right recursive `path/2` program from Fig. 2.3 to illustrate how programs are transformed and augmented with tabling primitives to implement the DRA tabling mechanism.

Each clause in the original definition of `path/2` becomes a clause for a new distinct

```
path(X,Z):- tabled_call(path(X,Z),Sid,Eval),
            ((Eval=1 -> dra_loop(Sid,Alt), path0(path(X,Z),Sid,Alt) ; fail)
            ;
            consume_answer(path(X,Z),Sid)).

path0(path(X,Z),Sid,1):- edge(X,Y), path(Y,Z), new_answer(path(X,Z),Sid,1).
path0(path(X,Z),Sid,2):- edge(X,Z), new_answer(path(X,Z),Sid,2).
path0(path(X,Z),Sid,3):- looping_state(Sid).
```

Figure 5.2: Program transformation for the right recursive `path/2` program

predicate, `path0/3` in the example, with 3 arguments. The first argument is the previous head clause; the second is the *subgoal id* representing the corresponding subgoal frame; and the third is the number of the clause regarding the textual order it appears on the program. A `new_answer/3` primitive is added to the end of each clause body in order to insert the answers in the table space. In addition, an extra `path0/3` clause is used to determine when, after exhausting all the matching clauses, we should move from the normal state to the looping state.

The `path/2` clause is maintained so that we can call it from other predicates without any change. The tabling primitive `tabled_call/3` in the body of `path/2` inserts/checks for the subgoal in the table space and returns the subgoal id (`Sid`) and a value (`Eval`) saying if the subgoal should be evaluated (cases where `Eval` is bound to 1) or not (cases where `Eval` is bound to 0). The `dra_loop/2` primitive controls the evaluation process and the `consume_answer/2` primitive implements the process of consuming answers one at a time. Both primitives are implemented as backtrackable predicates. The `Alt` argument in the `dra_loop/2` primitive is used to determine the number of the `path0/3` clause to be explored next. Note that, when we call a repeated subgoal or when the subgoal is already completed, we simply consume answers from the corresponding subgoal frame (cases where `Eval` is bound to 0). Otherwise, we first execute the `dra_loop/2` primitive and then, when it fails, we also execute the `consume_answer/2` primitive to consume answers. Next, we show an example that better illustrates how these primitives are used to control the evaluation.

5.3 An Evaluation Example

Figures 5.3 to 5.6 show the evaluation sequence for the query goal `p(1,Z)` if applying the program transformation for DRA evaluation presented in Fig. 5.2. At the top, each figure illustrates the program code and the state of the table space at the end of the evaluation represented in the figure. The bottom sub-figures show the resulting forest of trees with the numbering of nodes denoting the evaluation sequence. For illustration purposes the program code was simplified, predicates `consume_answer/2`, `path/2`, `path0/3` and `edge/2` are respectively denoted as `consume/2`, `p/2`, `p0/3` and `e/2`.

The evaluation starts with the `tabled_call/3` primitive being called for the `p(1,Z)` subgoal. As `p(1,Z)` is a first call (first calls are represented by black oval boxes), a new subgoal frame, with id `sid1`, is allocated in the table space for it and `Eval` is bound to 1 (step 2). The `dra_loop/2` primitive then starts exploring the first alternative of `p0/3` (`Alt` is bound to 1) and, in the continuation, `p(2,Z)` is first called (step 6).

As `p(2,Z)` is also a first call, we add a new subgoal frame, with id `sid2` (step 7), and proceed with `p(2,Z)` being resolved against the first clause for `p0/3` (step 9). In the continuation, subgoal `p(1,Z)` is called again (step 11). Since `p(1,Z)` is now a repeated call (repeated calls are represented by white oval boxes), we mark the alternatives in evaluation, up to the first call for `p(1,Z)`, as looping alternatives (step 12). This includes the first alternative for `p(2,Z)` and the first alternative for `p(1,Z)`.

Next, we try to consume answers (step 13) and because no answers are available for `p(1,Z)`, the `consume/2` primitive simply fails (step 14). The `dra_loop/2` primitive at node 8 then tries the next unexplored alternative of `p0/3` for `p(2,Z)`, alternative two in this case (step 15), and we obtain a first answer for `p(2,Z)` (step 17). The answer is inserted in the table and, because we are following a local scheduling strategy, the execution fails (step 18). We then backtrack again to the `dra_loop/2` primitive and now we try the last alternative which moves `p(2,Z)` to the looping state (step 20).

We have found a new answer for `p(2,Z)`, so we re-execute the looping alternative for `p(2,Z)` (step 22). The evaluation calls again `p(1,Z)`, but we fail because no answers are still available for `p(1,Z)` (step 27). The evaluation then backtracks to the `dra_loop/2` primitive and, because we have reached a partial fix-point, we check whether `p(2,Z)` can be completed. It can not, because it depends on `p(1,Z)`. We thus fail in order to consume the answer `p(2,1)` (step 29). The binding `Z=1` is then propagated to the context of subgoal `p(1,Z)`, and a first answer for `p(1,Z)` is found (step 30).

We then explore the second alternative of `p0/3` for `p(1,Z)` (step 32) obtaining, in the continuation, a second answer for `p(1,Z)` (step 34). The evaluation then proceeds with the third alternative of `p0/3` for `p(1,Z)`, which moves `p(1,Z)` from the normal state to the looping state (step 37).

```
p(X,Z):- tabled_call(p(X,Z),Sid,Eval),
         ((Eval=1 -> dra_loop(Sid,Alt), p0(p(X,Z),Sid,Alt) ; fail)
         ; consume(p(X,Z),Sid)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid,1).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid,2).
p0(p(X,Z),Sid,3):- looping_state(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Looping Alternatives
sid1	2. p(1,Z)		12. p0(p(1,Z),sid1,1)
sid2	7. p(2,Z)	17. p(2,1)	12. p0(p(2,Z),sid2,1)

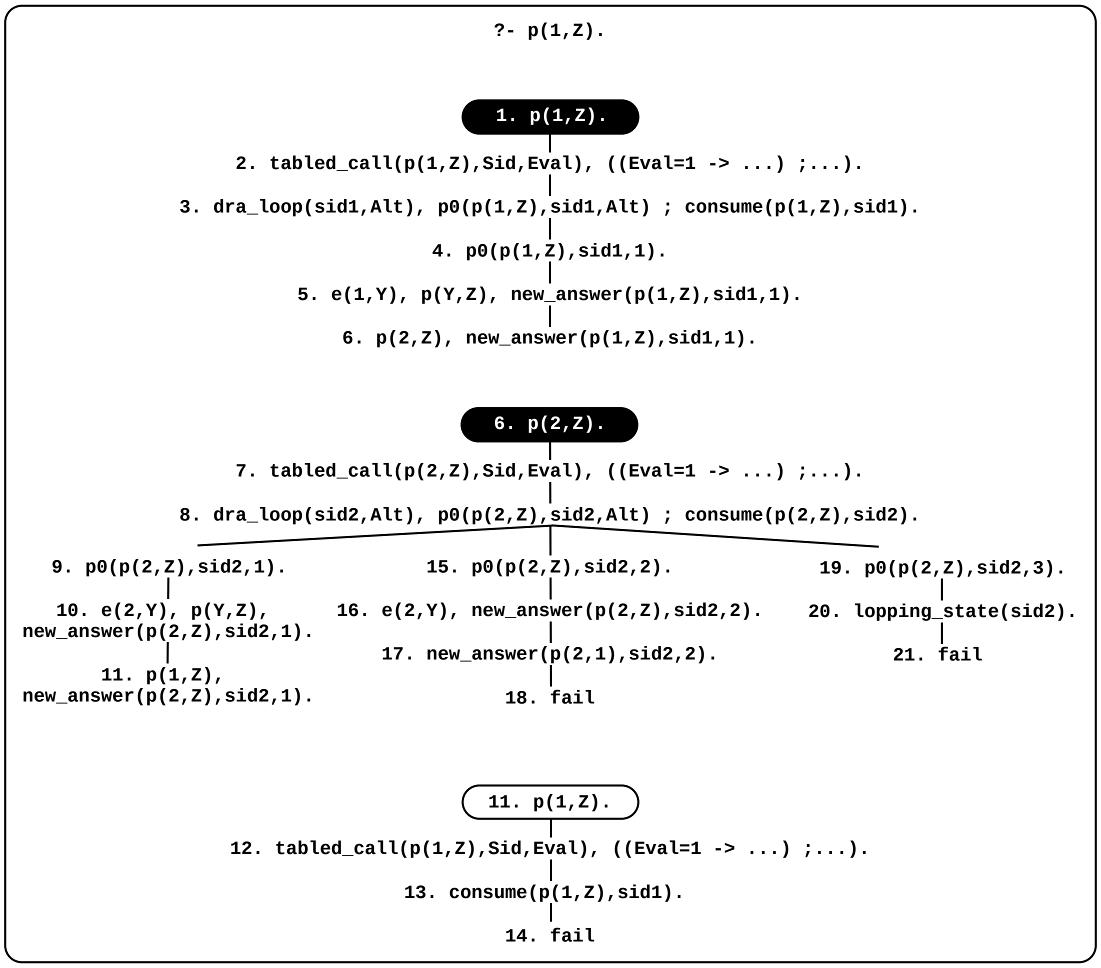

?- p(1,Z).
1. p(1,Z).
2. tabled_call(p(1,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
3. dra_loop(sid1,Alt), p0(p(1,Z),sid1,Alt) ; consume(p(1,Z),sid1).
4. p0(p(1,Z),sid1,1).
5. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1,1).
6. p(2,Z), new_answer(p(1,Z),sid1,1).
6. p(2,Z).
7. tabled_call(p(2,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
8. dra_loop(sid2,Alt), p0(p(2,Z),sid2,Alt) ; consume(p(2,Z),sid2).
9. p0(p(2,Z),sid2,1).
10. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2,1).
11. p(1,Z), new_answer(p(2,Z),sid2,1).
15. p0(p(2,Z),sid2,2).
16. e(2,Y), new_answer(p(2,Z),sid2,2).
17. new_answer(p(2,1),sid2,2).
18. fail
19. p0(p(2,Z),sid2,3).
20. lopping_state(sid2).
21. fail
11. p(1,Z).
12. tabled_call(p(1,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
13. consume(p(1,Z),sid1).
14. fail

Figure 5.3: DRA evaluation for the right recursive `path/2` program (steps 1 to 21)

Since we have found two new answers for `p(1,Z)`, the `dra_loop/2` primitive at node 3 then tries the looping alternative for `p(1,Z)` (step 39) which leads, in the continuation, to a new call to `p(2,Z)` (step 41). Because `p(2,Z)` has already reached the looping

```
p(X,Z):- tabled_call(p(X,Z),Sid,Eval),
         ((Eval=1 -> dra_loop(Sid,Alt), p0(p(X,Z),Sid,Alt) ; fail)
         ; consume(p(X,Z),Sid)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid,1).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid,2).
p0(p(X,Z),Sid,3):- looping_state(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Looping Alternatives
sid1	2. p(1,Z)	30. p(1,1) 34. p(1,2)	12. p0(p(1,Z),sid1,1)
sid2	7. p(2,Z)	17. p(2,1)	12. p0(p(2,Z),sid2,1)

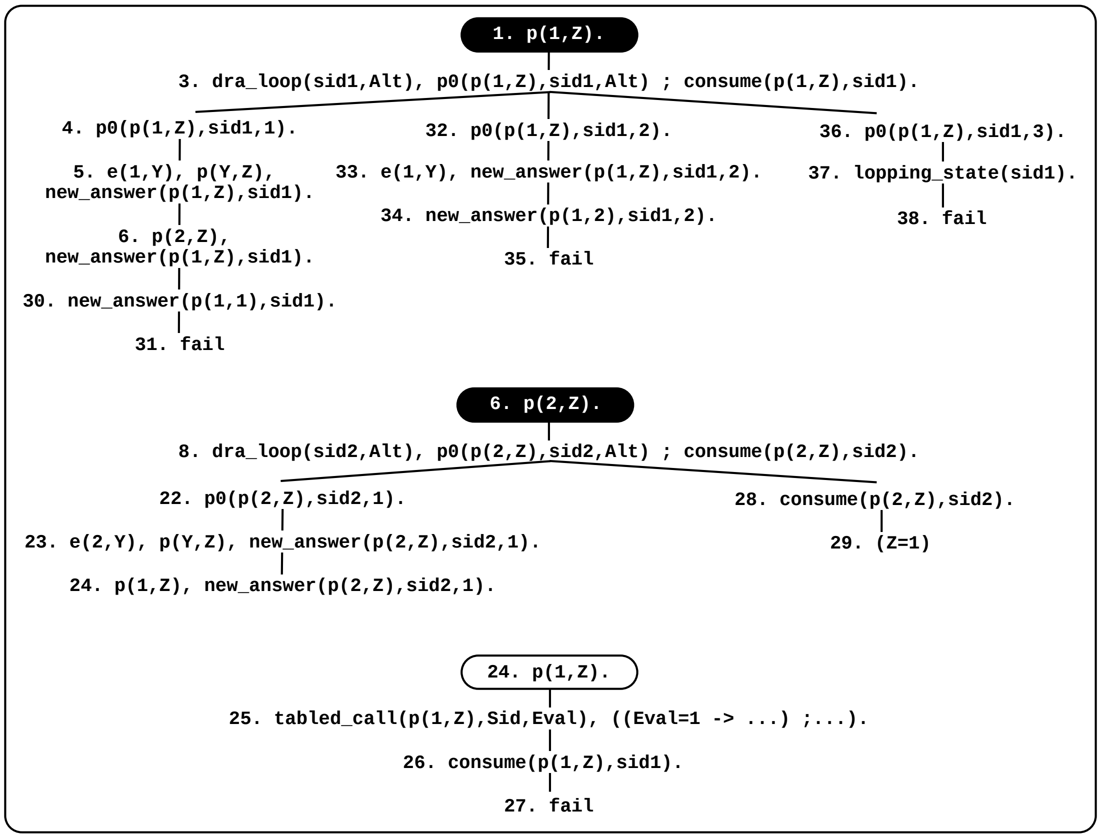

1. p(1,Z).
3. dra_loop(sid1,Alt), p0(p(1,Z),sid1,Alt) ; consume(p(1,Z),sid1).
4. p0(p(1,Z),sid1,1).
5. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1).
6. p(2,Z), new_answer(p(1,Z),sid1).
30. new_answer(p(1,1),sid1).
31. fail
32. p0(p(1,Z),sid1,2).
33. e(1,Y), new_answer(p(1,Z),sid1,2).
34. new_answer(p(1,2),sid1,2).
35. fail
36. p0(p(1,Z),sid1,3).
37. lopping_state(sid1).
38. fail
6. p(2,Z).
8. dra_loop(sid2,Alt), p0(p(2,Z),sid2,Alt) ; consume(p(2,Z),sid2).
22. p0(p(2,Z),sid2,1).
23. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2,1).
24. p(1,Z), new_answer(p(2,Z),sid2,1).
28. consume(p(2,Z),sid2).
29. (Z=1)
24. p(1,Z).
25. tabled_call(p(1,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
26. consume(p(1,Z),sid1).
27. fail

Figure 5.4: DRA evaluation for the right recursive `path/2` program (steps 22 to 38)

state, we proceed with `p(2,Z)` being resolved against its looping alternative (step 44). Subgoal `p(1,Z)` is then called again (step 46) and we consume the two available answers for it. The bindings `Z=1` (step 49) and `Z=2` (step 52) are propagated to the context of subgoal `p(2,Z)`, but only `Z=2` produces a new answer for `p(2,Z)`, `p(2,2)` in this case (step 53).

Backtracking then sends us back to the `dra_loop/2` primitive at node 43 and because

```
p(X,Z):- tabled_call(p(X,Z),Sid,Eval),
         ((Eval=1 -> dra_loop(Sid,Alt), p0(p(X,Z),Sid,Alt) ; fail)
         ; consume(p(X,Z),Sid)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid,1).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid,2).
p0(p(X,Z),Sid,3):- looping_state(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Looping Alternatives
sid1	2. p(1,Z)	30. p(1,1) 34. p(1,2)	12. p0(p(1,Z),sid1,1)
sid2	7. p(2,Z)	17. p(2,1) 53. p(2,2)	12. p0(p(2,Z),sid2,1)

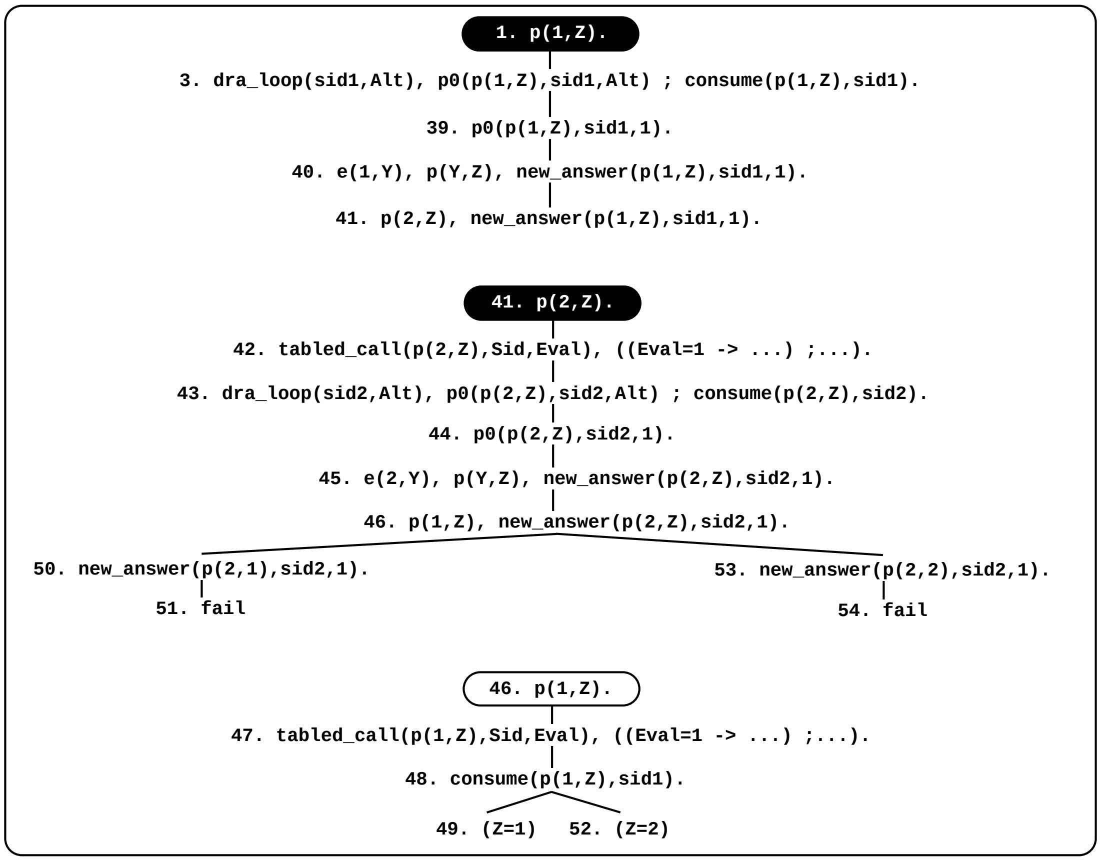

1. p(1,Z).
3. dra_loop(sid1,Alt), p0(p(1,Z),sid1,Alt) ; consume(p(1,Z),sid1).
39. p0(p(1,Z),sid1,1).
40. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1,1).
41. p(2,Z), new_answer(p(1,Z),sid1,1).
41. p(2,Z).
42. tabled_call(p(2,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
43. dra_loop(sid2,Alt), p0(p(2,Z),sid2,Alt) ; consume(p(2,Z),sid2).
44. p0(p(2,Z),sid2,1).
45. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2,1).
46. p(1,Z), new_answer(p(2,Z),sid2,1).
50. new_answer(p(2,1),sid2,1).
51. fail
53. new_answer(p(2,2),sid2,1).
54. fail
46. p(1,Z).
47. tabled_call(p(1,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
48. consume(p(1,Z),sid1).
49. (Z=1)
52. (Z=2)

Figure 5.5: DRA evaluation for the right recursive `path/2` program (steps 39 to 54)

we have found a new answer for `p(2,Z)`, we re-execute the looping alternative (step 55). Next, we call again `p(1,Z)`, but this time only redundant answers are found (steps 59 to 65). We thus fail in order to propagate the answers for `p(2,Z)` to `p(1,Z)`, but no new answers are found for `p(1,Z)` (steps 66 to 72). The evaluation then backtracks to the `dra_loop/2` primitive for `p(1,Z)` and, because subgoals `p(1,Z)` and `p(2,Z)` are now fully exploited, we can declared the two subgoals to be completed (step 73).

```
p(X,Z):- tabled_call(p(X,Z),Sid,Eval),
         ((Eval=1 -> dra_loop(Sid,Alt), p0(p(X,Z),Sid,Alt) ; fail)
         ; consume(p(X,Z),Sid)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid,1).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid,2).
p0(p(X,Z),Sid,3):- looping_state(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Looping Alternatives
sid1	2. p(1,Z)	30. p(1,1) 34. p(1,2) 73. complete	12. p0(p(1,Z),sid1,1)
sid2	7. p(2,Z)	17. p(2,1) 53. p(2,2) 73. complete	12. p0(p(2,Z),sid2,1)

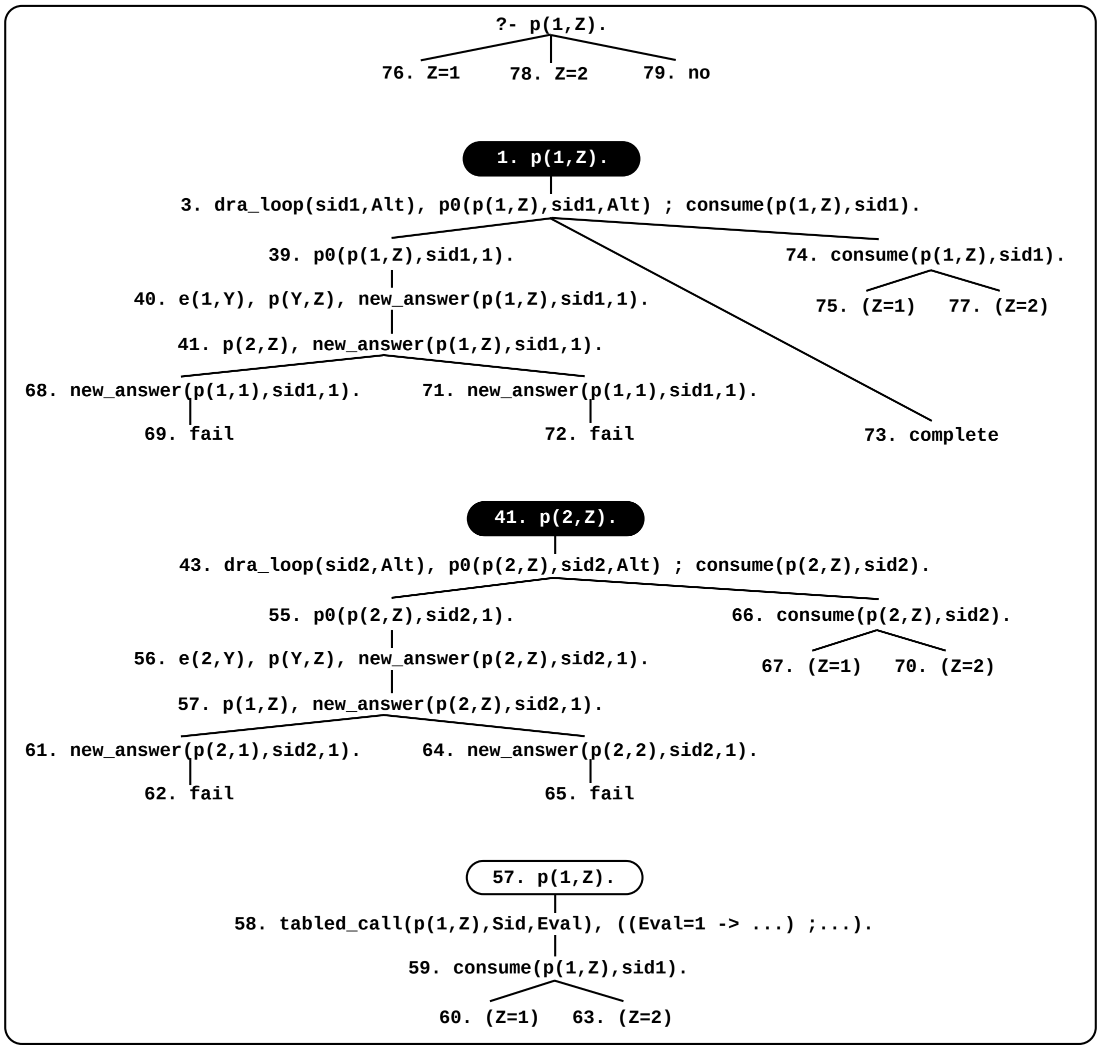

?- p(1,Z).
76. Z=1
78. Z=2
79. no
1. p(1,Z).
3. dra_loop(sid1,Alt), p0(p(1,Z),sid1,Alt) ; consume(p(1,Z),sid1).
39. p0(p(1,Z),sid1,1).
74. consume(p(1,Z),sid1).
40. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1,1).
75. (Z=1)
77. (Z=2)
41. p(2,Z), new_answer(p(1,Z),sid1,1).
68. new_answer(p(1,1),sid1,1).
71. new_answer(p(1,1),sid1,1).
69. fail
72. fail
73. complete
41. p(2,Z).
43. dra_loop(sid2,Alt), p0(p(2,Z),sid2,Alt) ; consume(p(2,Z),sid2).
55. p0(p(2,Z),sid2,1).
66. consume(p(2,Z),sid2).
56. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2,1).
67. (Z=1)
70. (Z=2)
57. p(1,Z), new_answer(p(2,Z),sid2,1).
61. new_answer(p(2,1),sid2,1).
64. new_answer(p(2,2),sid2,1).
62. fail
65. fail
57. p(1,Z).
58. tabled_call(p(1,Z),Sid,Eval), ((Eval=1 -> ...) ;...).
59. consume(p(1,Z),sid1).
60. (Z=1)
63. (Z=2)

Figure 5.6: DRA evaluation for the right recursive `path/2` program (steps 55 to 79)

Next, we consume the two available answers for `p(1,Z)` obtaining, in the continuation, the two answers for the query goal (steps 76 and 78). Finally, at step 79, we return *no* to the query goal.

5.4 Implementation Details

We next describe in more detail all the implementation issues required to fully support the DRA linear tabling mechanism.

5.4.1 Subgoal Frames

In the DRA linear tabling mechanism, a subgoal frame is an eleven field data structure. The `SgFr_state`, `SgFr_answers`, `SgFr_first_answer`, `SgFr_last_answer`, `SgFr_dfn`, `SgFr_dep` and `SgFr_previous` fields are used as for the tabled evaluation with continuation calls mechanism. The four extra fields have the following meaning:

- `SgFr_current_alt:` is the number of the alternative being evaluated.
- `SgFr_new_answers:` indicates if new answers were found during the normal state or during the execution of the last looping alternative, if in the looping state.
- `SgFr_last_alt:` is the number of the alternative where we have found the last answer.
- `SgFr_looping_alts:` is the pointer to the looping alternatives associated with the subgoal or `NULL` if no looping alternatives are available.

The corresponding pseudo-code for the `new_subgoal_frame()` procedure is presented next in Fig. 5.7. Figure 5.8 uses the example from Figures 5.3 to 5.6 to illustrate how these fields are used and updated during a tabled evaluation.

Figure 5.8(a) shows the state of the table space after calling the `tabled_call/3` primitive for the `p(1,Z)` subgoal (step 2 in Fig. 5.3). As `p(1,Z)` is a first call, the `tabled_call/3` primitive allocates and initializes a new subgoal frame in the table space for it. The `dra_loop/2` primitive then starts exploring the first alternative of `p0/3` for `p(1,Z)` and the `SgFr_current_alt` field of the subgoal frame for `p(1,Z)` is updated to 1. In the continuation, the `tabled_call/3` primitive is called for the `p(2,Z)` subgoal and a new subgoal frame is also allocated and initialized (step 7 in Fig. 5.3). Figure 5.8(b) shows the resulting state of both subgoal frames at that point.

```
new_subgoal_frame() {
  sf = allocate_subgoal_frame();
  SgFr_state(sf) = READY;
  SgFr_answers(sf) = open_trie();
  SgFr_first_answer(sf) = NULL;
  SgFr_last_answer(sf) = NULL;
  SgFr_current_alt(sf) = 0;
  SgFr_new_answers(sf) = FALSE;
  SgFr_last_alt(sf) = 0;
  SgFr_looping_alts(sf) = NULL;
  SgFr_previous(sf) = SF_TOP;
  if (SF_TOP == NULL)
    SgFr_dfn(sf) = 1;
  else
    SgFr_dfn(sf) = SgFr_dfn(SF_TOP) + 1;
  SgFr_dep(sf) = SgFr_dfn(sf);
  SF_TOP = sf;
  return sf;
}
```

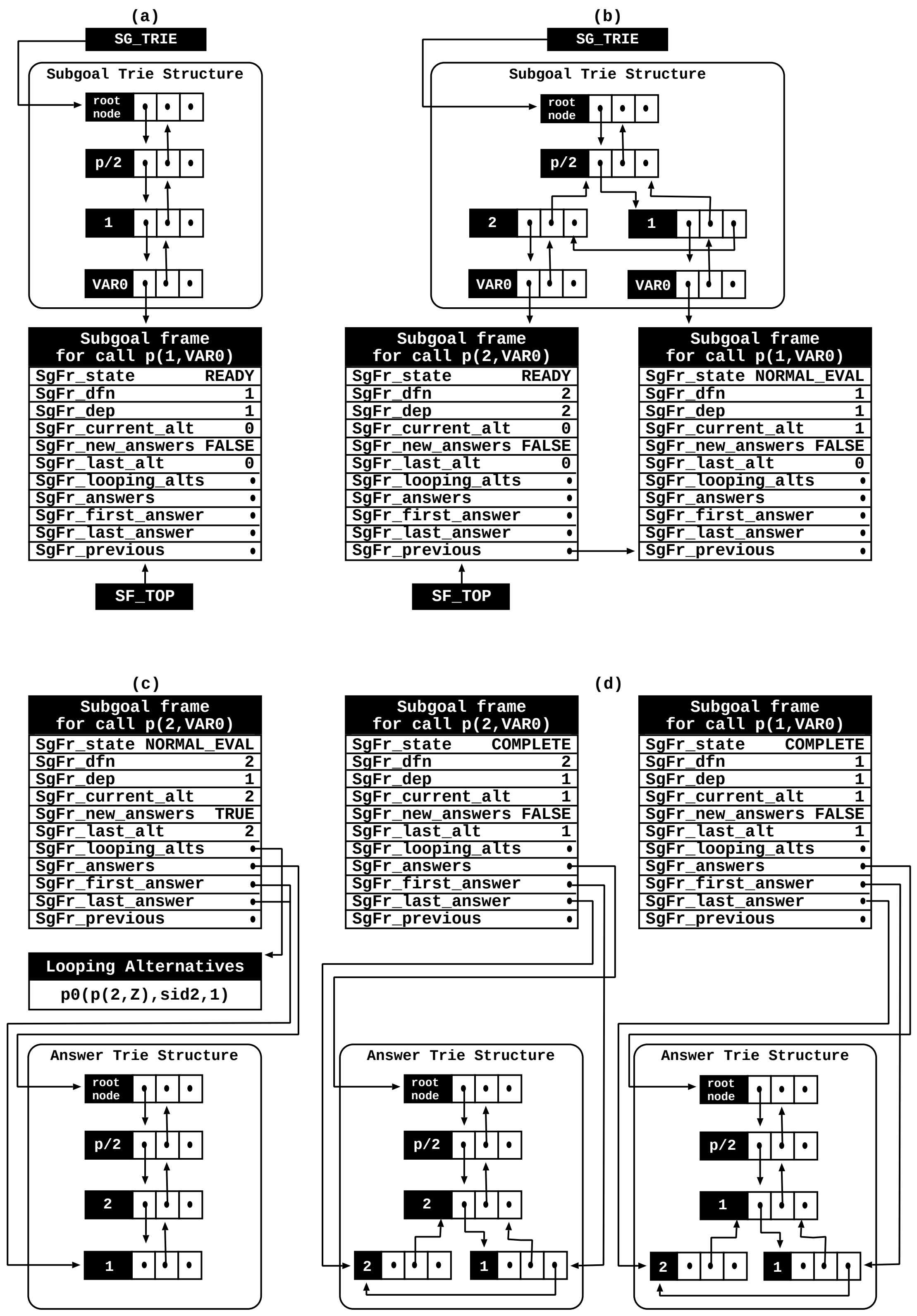
Figure 5.7: Pseudo-code for `new_subgoal_frame()`

Next, subgoal `p(1,Z)` is called again (step 11 in Fig. 5.3). The alternatives in evaluation for `p(2,Z)` and `p(1,Z)` are marked as looping alternatives and the `SgFr_dep` field of the subgoal frame for `p(2,Z)` is updated to 1, the depth-first number of `p(1,Z)`. We then try the second alternative for `p(2,Z)` (step 15 in Fig. 5.3) and, in the continuation, we obtain a first answer for `p(2,Z)` (step 17 in Fig. 5.3). The answer is inserted in the answer trie for `p(2,Z)`, the `SgFr_new_answers` field of the subgoal frame for `p(2,Z)` is updated to `TRUE` and the `SgFr_last_alt` field is updated to 2, the number of the alternative in evaluation. Figure 5.8(c) shows the resulting state of the subgoal frame for `p(2,Z)` at that point.

The evaluation then obtains two answers for `p(1,Z)` (steps 30 and 34 in Fig. 5.4) and a second answer for `p(2,Z)` (step 53 in Fig. 5.5). At the end, both subgoals are declared as complete (step 73 in Fig. 5.6). This includes marking the corresponding subgoal frames as complete, deleting the looping alternatives, and updating the `SF_TOP` variable to `NULL`. Figure 5.8(d) shows the final state of both subgoal frames.

5.4.2 Tabling Primitives

We next show the implementation details for the five tabling primitives that support the DRA linear tabling mechanism. We start with Fig. 5.9 showing the pseudo-code for the `new_answer/3` primitive.



Subgoal frame for call p(1,VAR0)	(a)
SgFr_state	READY
SgFr_dfn	1
SgFr_dep	1
SgFr_current_alt	0
SgFr_new_answers	FALSE
SgFr_last_alt	0
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Subgoal frame for call p(2,VAR0)	(b)
SgFr_state	READY
SgFr_dfn	2
SgFr_dep	2
SgFr_current_alt	0
SgFr_new_answers	FALSE
SgFr_last_alt	0
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Subgoal frame for call p(1,VAR0)	(b)
SgFr_state	NORMAL_EVAL
SgFr_dfn	1
SgFr_dep	1
SgFr_current_alt	1
SgFr_new_answers	FALSE
SgFr_last_alt	0
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Subgoal frame for call p(2,VAR0)	(c)
SgFr_state	NORMAL_EVAL
SgFr_dfn	2
SgFr_dep	1
SgFr_current_alt	2
SgFr_new_answers	TRUE
SgFr_last_alt	2
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Subgoal frame for call p(2,VAR0)	(d)
SgFr_state	COMPLETE
SgFr_dfn	2
SgFr_dep	1
SgFr_current_alt	1
SgFr_new_answers	FALSE
SgFr_last_alt	1
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Subgoal frame for call p(1,VAR0)	(d)
SgFr_state	COMPLETE
SgFr_dfn	1
SgFr_dep	1
SgFr_current_alt	1
SgFr_new_answers	FALSE
SgFr_last_alt	1
SgFr_looping_alts	•
SgFr_answers	•
SgFr_first_answer	•
SgFr_last_answer	•
SgFr_previous	•

Figure 5.8: Subgoal frames at different points of the evaluation of the right recursive `path/2` program of Figures 5.3 to 5.6

```
new_answer(YAP_Term ANSWER, SgFr SF, YAP_Int ALT) {
  if (put_answer(SF, ANSWER) == TRUE) {
    SgFr_new_answers(SF) = TRUE;
    SgFr_last_alt(SF) = ALT;
  }
  return FALSE;                                        // always fail at the end
}
```

Figure 5.9: Pseudo-code for the `new_answer/3` primitive

The `new_answer/3` primitive calls the `put_answer()` procedure to insert the given `ANSWER` in the answer trie structure for the `SF` subgoal frame and, if the answer is new, it updates the `SgFr_new_answers` to `TRUE` and the `SgFr_last_alt` to the alternative in evaluation, as given by the `ALT` argument. We then implement a local scheduling approach and always fail at the end.

When we reach the last clause of a tabled predicate, we execute the `looping_state/1` primitive. Figure 5.10 shows the pseudo-code for it.

```
looping_state(SgFr SF) {
  if (SgFr_new_answers(SF) == TRUE)
    SgFr_state(SF) = LOOPING;                       // move to the looping state
  else
    SgFr_state(SF) = NO_LOOPING;
  return FALSE;                                        // always fail at the end
}
```

Figure 5.10: Pseudo-code for the `looping_state/1` primitive

The `looping_state/1` primitive simply checks if any answer was found during the normal state. If so (cases where `SgFr_new_answers` is `TRUE`), it changes the subgoal's state to `LOOPING`. Otherwise, it updates the subgoal's state to `NO_LOOPING`. We then implement a local scheduling approach and always fail at the end. The computation then returns to the `dra_loop/2` primitive for the tabled subgoal call in execution, and the `LOOPING` and `NO_LOOPING` states are then used to decide whether it should start executing the looping alternatives, fail or complete.

We next show in Fig. 5.11 the pseudo-code for the `tabled_call/3` primitive. As for the tabled evaluation with continuation calls mechanism, the `tabled_call/3` primitive starts by calling the `put_tabled_call()` procedure in order to insert the given `SUBGOAL_CALL` in the subgoal trie structure. Then, if the resulting subgoal frame is new, that is, if the `SgFr_state` is `READY`, it changes the subgoal's state to `NORMAL_EVAL` and unifies the `EVAL` argument with 1.

```
tabled_call(YAP_Term SUBGOAL_CALL, SgFr SF, YAP_Int EVAL) {
  sf = put_tabled_call(SUBGOAL_CALL);
  YAP_Unify(SF, sf);
  if (SgFr_state(sf) == READY) {
    SgFr_state(sf) = NORMAL_EVAL;
    YAP_Unify(EVAL, 1);                    // go to the dra_loop/2 primitive
    return TRUE;
  }
  if (SgFr_state(sf) == LOOPING) {
    YAP_Unify(EVAL, 1);                    // go to the dra_loop/2 primitive
    return TRUE;
  }
  if (SgFr_state(sf) != COMPLETE) {    // propagate dependencies and add ...
    sf_aux = SF_TOP;                   //  ... looping alternatives up to sf
    while (sf != sf_aux) {
      SgFr_dep(sf_aux) = minimum(SgFr_dep(sf_aux), SgFr_dep(sf));
      if (SgFr_state(sf) == NORMAL_EVAL)
        add_looping_alternative(sf_aux);
      sf_aux = SgFr_previous(sf_aux);
    }
    if (SgFr_state(sf) == NORMAL_EVAL)
      add_looping_alternative(sf);
  }
  YAP_Unify(EVAL, 0);               // go to the consume_answer/2 primitive
  return TRUE;
}
```

Figure 5.11: Pseudo-code for the `tabled_call/3` primitive

If the resulting subgoal frame is not new then the `SgFr_state` can be in one of the following states: `NORMAL_EVAL`, `LOOPING`, `LOOPING_EVAL` or `COMPLETE`. When a subgoal executes the `looping_state/1` primitive to move from the normal state to the looping state, the `SgFr_state` is first marked as `LOOPING`. Then, when the subgoal starts evaluating its looping alternatives, it enters the `LOOPING_EVAL` state, returning to the `LOOPING` state when it reaches a partial fix-point for the looping alternatives. The `LOOPING_EVAL`/`LOOPING` sequence can be repeated several times until the subgoal be marked as `COMPLETE`. Thus, if after calling the `put_tabled_call()` procedure, the `SgFr_state` of the resulting subgoal frame is in the `LOOPING` state, then we simply unify the `EVAL` argument with 1 and succeed. This is the situation that occurs at step 42 in Fig. 5.5 with the `tabled_call/3` primitive for the `p(2,Z)` subgoal.

Otherwise, if the subgoal is already in evaluation, that is, if the `SgFr_state` is marked as `NORMAL_EVAL` or `LOOPING_EVAL`, we propagate the dependency in the `SgFr_dep` field of the current subgoal to the `SgFr_dep` field of all younger subgoal frames. Moreover, if the current subgoal is in the normal state, we mark the alternatives in evaluation as looping alternatives. When the subgoal is marked as `NORMAL_EVAL`, `LOOPING_EVAL` or

`COMPLETE` we should consume answers and for that we thus unify the `EVAL` argument with 0 and succeed. The `consume_answer/2` primitive for the DRA mechanism is the same as for the tabled evaluation with continuation calls mechanism (please refer to Fig. 4.10).

The `dra_loop/2` primitive controls the evaluation process. It was implemented in Yap as a backtrackable predicate. This is a special case of a backtrackable predicate because it uses the same procedure, the `dra_loop()` procedure, to start the execution of the primitive and to continue its execution when backtracking occurs. Figure 5.12 shows its implementation.

Initially, the procedure checks if the computation is in the normal state and, if so, it succeeds with the `ALT` argument bound to the number of the next clause to be explored. Otherwise, it checks if the subgoal's state is `LOOPING` and, if there are looping alternatives, it gets the next looping alternative, changes the subgoal's state to `LOOPING_EVAL`, resets the `SgFr_last_alt` and `SgFr_new_answers` fields and succeeds with the `ALT` argument bound to the number of the looping alternative.

Then, if the current subgoal is already in the looping state and being evaluated (cases where the `SgFr_state` is `LOOPING_EVAL`), then it gets the next looping alternative, resets the `SgFr_new_answers` field and succeeds with the `ALT` argument bound to the number of the next looping alternative (when we have a single looping alternative, the next looping alternative is always the same). This process repeats until we have reached a partial fix-point, that is, until no new answers were found during one cycle of trying the looping alternatives. In the continuation, it checks for completion and if the `SgFr_dfn` and `SgFr_dep` fields are equal then no dependencies exist and, therefore, all younger subgoals can be completed. Otherwise, the subgoal returns to the `LOOPING` state.

5.4.3 Detecting Completion

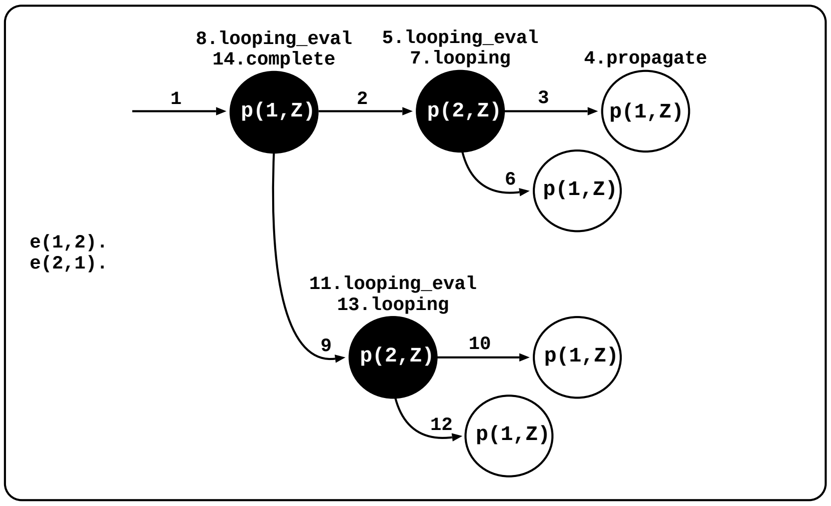
In the DRA execution model, completion is detected in the `dra_loop/2` primitive. As for the tabled evaluation with continuation calls mechanism, we use the `SgFr_dfn` and the `SgFr_dep` fields of the subgoal frames to quickly determine whether a subgoal is a leader node. Moreover, we use the `SgFr_state`, `SgFr_new_answers`, `SgFr_current_alt` and `SgFr_last_alt` fields of the subgoal frames to determine when we reach a partial fix-point for the looping alternatives. Figure 5.13 uses again the example from Figures 5.3 to 5.6 to illustrate how completion is detected in the DRA execution model

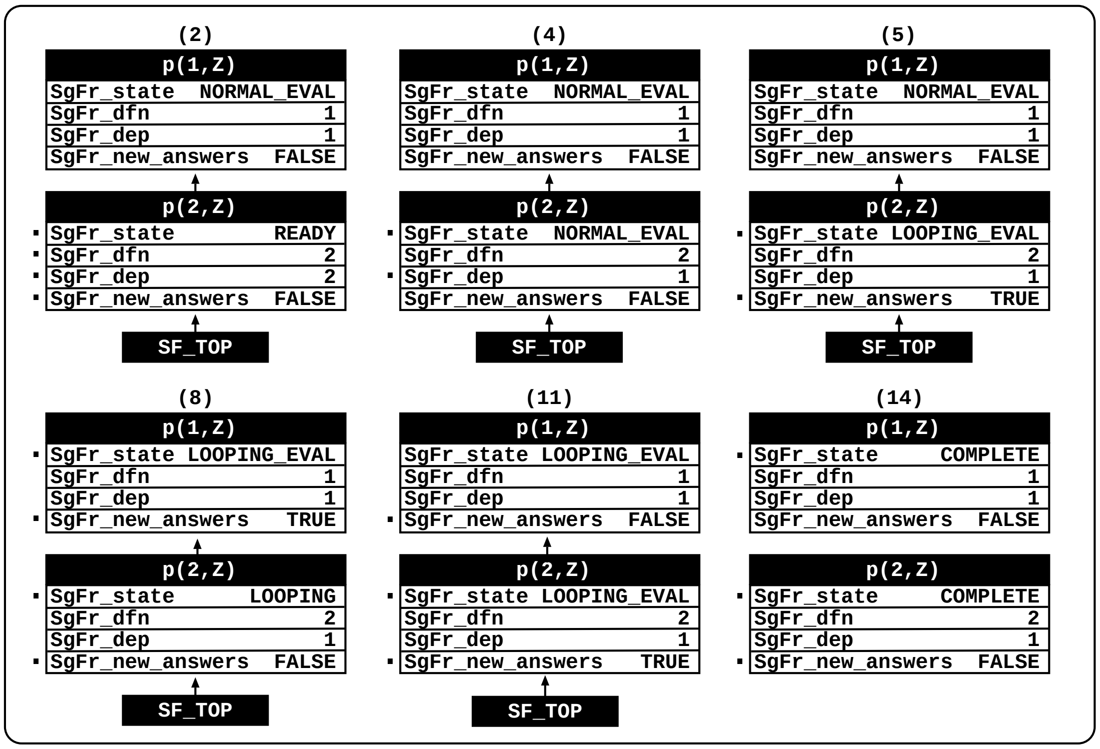
```
dra_loop(SgFr SF, YAP_Int ALT) {
  if (SgFr_state(SF) == NORMAL_EVAL) {
    SgFr_current_alt(SF)++;
    YAP_Unify(ALT, SgFr_current_alt(SF));
    return TRUE;
  }
  if (SgFr_state(SF) == LOOPING) {
    SgFr_current_alt(SF) = get_next_looping_alternative(SF);
    if (SgFr_current_alt(SF) == 0) {              // no looping alternatives
      YAP_cut_fail();
      return FALSE;
    }
    SgFr_state(SF) = LOOPING_EVAL;
    SgFr_last_alt(SF) = SgFr_current_alt(SF);
    SgFr_new_answers(SF) = FALSE;
    YAP_Unify(ALT, SgFr_current_alt(SF));
    return TRUE;
  }
  if (SgFr_state(SF) == LOOPING_EVAL) {
    SgFr_current_alt(SF) = get_next_looping_alternative(SF);
    if (SgFr_last_alt(SF) != SgFr_current_alt(SF) || SgFr_new_answers(SF)) {
      SgFr_new_answers(SF) = FALSE;
      YAP_Unify(ALT, SgFr_current_alt(SF));
      return TRUE;
    }
  }
  if (SgFr_dfn(SF) == SgFr_dep(SF)) {        // complete all frames up to SF
    while (SF_TOP != SF) {
      SgFr_state(SF_TOP) = COMPLETE;
      delete_looping_alternatives(SF_TOP);
      SF_TOP = SgFr_previous(SF_TOP);
    }
    SgFr_state(SF) = COMPLETE;
    delete_looping_alternatives(SF);
  } else
    SgFr_state(SF) = LOOPING;
  YAP_cut_fail();
  return FALSE;                      // go to the consume_answer/2 primitive
}
```

Figure 5.12: Pseudo-code for the `dra_loop/2` primitive

(we omit the `SgFr_current_alt` and `SgFr_last_alt` fields because we have only a single looping alternative). The black dots in the sub-figure below indicates the fields being updated at each step of the example.

Initially, `p(1,Z)` and `p(2,Z)` are first called and two subgoal frames are allocated (step 2 in Fig. 5.13). In the continuation, subgoal `p(1,Z)` is called again, creating a dependency between subgoals `p(2,Z)` and `p(1,Z)`, and the `SgFr_dep` field of subgoal `p(2,Z)` is updated to represent this dependency (step 4 in Fig. 5.13). The execution





(2)

	p(1,Z)	
	SgFr_state	NORMAL_EVAL
	SgFr_dfn	1
	SgFr_dep	1
	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	READY
▪	SgFr_dfn	2
▪	SgFr_dep	2
▪	SgFr_new_answers	FALSE

SF_TOP

(4)

	p(1,Z)	
	SgFr_state	NORMAL_EVAL
	SgFr_dfn	1
	SgFr_dep	1
	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	NORMAL_EVAL
	SgFr_dfn	2
▪	SgFr_dep	1
	SgFr_new_answers	FALSE

SF_TOP

(5)

	p(1,Z)	
	SgFr_state	NORMAL_EVAL
	SgFr_dfn	1
	SgFr_dep	1
	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	LOOPING_EVAL
	SgFr_dfn	2
	SgFr_dep	1
▪	SgFr_new_answers	TRUE

SF_TOP

(8)

	p(1,Z)	
▪	SgFr_state	LOOPING_EVAL
	SgFr_dfn	1
	SgFr_dep	1
▪	SgFr_new_answers	TRUE

	p(2,Z)	
▪	SgFr_state	LOOPING
	SgFr_dfn	2
	SgFr_dep	1
▪	SgFr_new_answers	FALSE

SF_TOP

(11)

	p(1,Z)	
	SgFr_state	LOOPING_EVAL
	SgFr_dfn	1
	SgFr_dep	1
▪	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	LOOPING_EVAL
	SgFr_dfn	2
	SgFr_dep	1
▪	SgFr_new_answers	TRUE

SF_TOP

(14)

	p(1,Z)	
▪	SgFr_state	COMPLETE
	SgFr_dfn	1
	SgFr_dep	1
	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	COMPLETE
	SgFr_dfn	2
	SgFr_dep	1
▪	SgFr_new_answers	FALSE

Figure 5.13: Detecting completion when evaluating the right recursive `path/2` program of Figures 5.3 to 5.6

then finds a first answer for `p(2,Z)` and the `SgFr_new_answers` field of the subgoal frame for `p(2,Z)` is updated to `TRUE`. Next, subgoal `p(2,Z)` moves to the looping state and we re-execute its looping alternative because we have found an answer for it (step 5 in Fig. 5.13).

The evaluation then calls again `p(1,Z)` but we fail because no answers are still available for it (step 6 in Fig. 5.13). Then, the computation returns to `p(2,Z)` but we cannot perform completion because the `SgFr_dfn` and `SgFr_dep` fields of `p(2,Z)` are different. We thus mark the subgoal as `LOOPING` and fail (step 7 in Fig. 5.13).

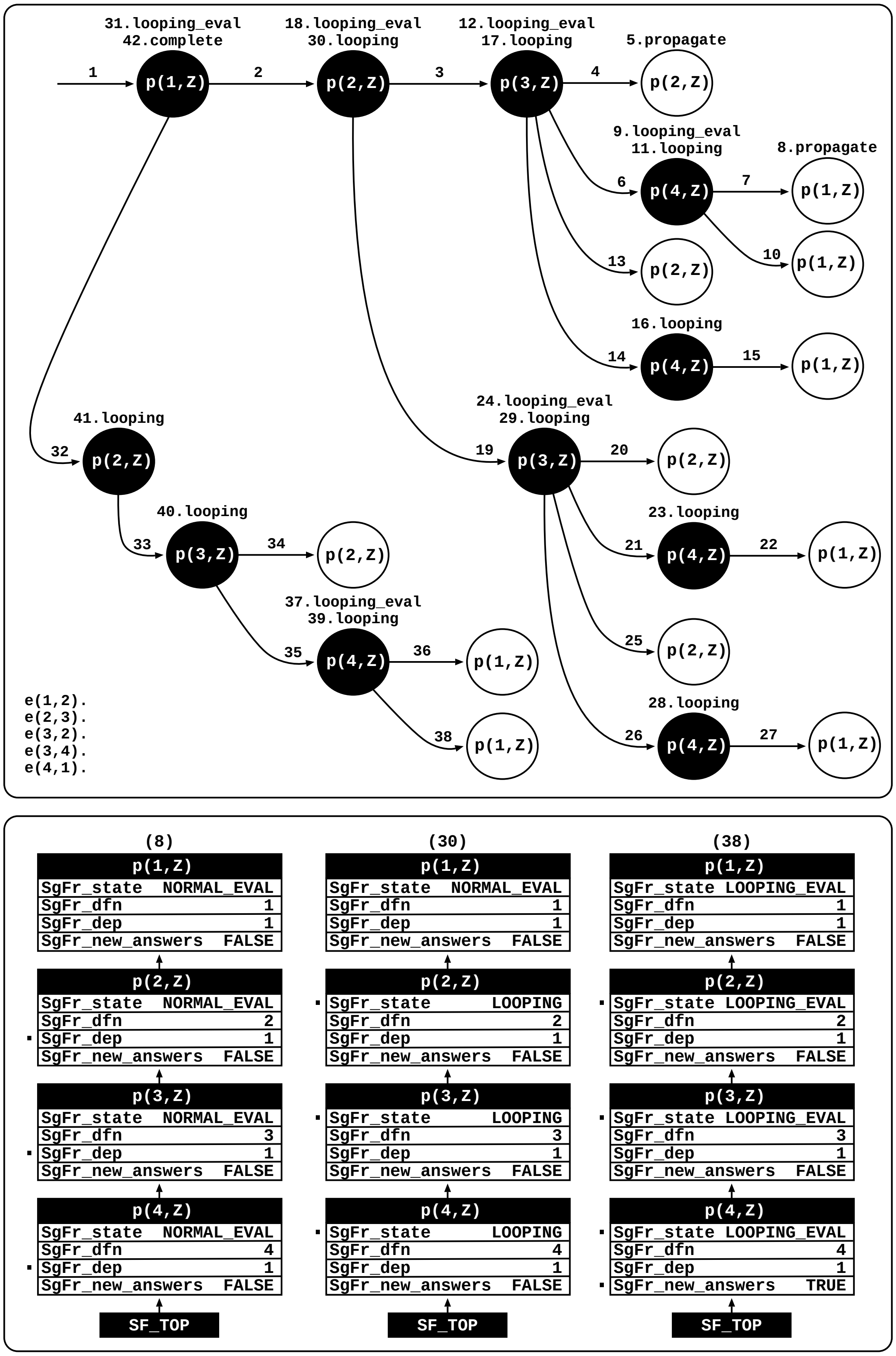
A similar situation then occurs with `p(1,Z)`. First, the execution finds two answers for `p(1,Z)` and its `SgFr_new_answers` field is updated to `TRUE`. Next, `p(1,Z)` moves to the looping state and we re-execute its looping alternative because we have found two answers for it (step 8 in Fig. 5.13).

The execution then proceeds with subgoals `p(2,Z)` and `p(1,Z)` being called again (steps 9 and 10 in Fig. 5.13) and, in the continuation, we find a new answer for `p(2,Z)`. Thus, when the computation returns to `p(2,Z)` we re-execute its looping alternative because we have found a new answer for it (step 11 in Fig. 5.13). Next, we call again `p(1,Z)`, but this time only redundant answers are found (step 12 in Fig. 5.13). The evaluation then returns first to `p(2,Z)` and then to `p(1,Z)` and, because subgoals `p(1,Z)` and `p(2,Z)` are now fully exploited, both subgoals are marked as `COMPLETE` (step 14 in Fig. 5.13).

Figure 5.14 shows an example for a more complex graph as represented by the `e/2` facts in the sub-figure at the top. Similarly to the previous example, we start by allocating and initializing frames to subgoals `p(1,Z)`, `p(2,Z)` and `p(3,Z)`. Next, `p(3,Z)` calls `p(2,Z)` and the `SgFr_dep` field of subgoal `p(3,Z)` is updated to represent this dependency (step 5 in Fig. 5.14). Next, `p(4,Z)` is also called and a new subgoal frame is allocated (step 6 in Fig. 5.14). In the continuation, `p(4,Z)` calls `p(1,Z)` again, and the `SgFr_dep` field of all subgoals up to `p(1,Z)` are updated to represent this new dependency (step 8 in Fig. 5.14).

The execution then finds a first answer for `p(4,Z)`, `p(4,1)`, which will force the re-execution of its looping alternative (step 9 Fig. 5.14). Subgoal `p(1,Z)` is then called again but we fail because no answers are still available for it (step 10 in Fig. 5.14). When backtracking to `p(4,Z)` we cannot perform completion because the current leader node is `p(1,Z)`. We thus mark `p(4,Z)` as `LOOPING` and fail (step 11 in Fig. 5.14). We then find three answers for `p(3,Z)`: `p(3,1)`, `p(3,2)` and `p(3,4)`, which will also force the re-execution of its looping alternative (step 12 in Fig. 5.14). Subgoals `p(2,Z)`, `p(4,Z)` and `p(1,Z)` are then called again but no new answers are found. In the continuation, `p(3,Z)` is also marked as `LOOPING` and the execution fails (step 17 in Fig. 5.14).

A similar situation then happens for `p(2,Z)`, the execution finds four answers for it: `p(2,1)`, `p(2,2)`, `p(2,4)` and `p(2,3)`, which will force the re-execution of its looping alternative (step 18 in Fig. 5.14). In the continuation, we find a new answer for `p(3,Z)`,



(8)

p(1,Z)	
SgFr_state	NORMAL_EVAL
SgFr_dfn	1
SgFr_dep	1
SgFr_new_answers	FALSE

p(2,Z)	
SgFr_state	NORMAL_EVAL
SgFr_dfn	2
▪ SgFr_dep	1
SgFr_new_answers	FALSE

p(3,Z)	
SgFr_state	NORMAL_EVAL
SgFr_dfn	3
▪ SgFr_dep	1
SgFr_new_answers	FALSE

p(4,Z)	
SgFr_state	NORMAL_EVAL
SgFr_dfn	4
▪ SgFr_dep	1
SgFr_new_answers	FALSE

(30)

p(1,Z)	
SgFr_state	NORMAL_EVAL
SgFr_dfn	1
SgFr_dep	1
SgFr_new_answers	FALSE

p(2,Z)	
▪ SgFr_state	LOOPING
SgFr_dfn	2
SgFr_dep	1
SgFr_new_answers	FALSE

p(3,Z)	
▪ SgFr_state	LOOPING
SgFr_dfn	3
SgFr_dep	1
SgFr_new_answers	FALSE

p(4,Z)	
▪ SgFr_state	LOOPING
SgFr_dfn	4
SgFr_dep	1
SgFr_new_answers	FALSE

(38)

p(1,Z)	
SgFr_state	LOOPING_EVAL
SgFr_dfn	1
SgFr_dep	1
SgFr_new_answers	FALSE

p(2,Z)	
▪ SgFr_state	LOOPING_EVAL
SgFr_dfn	2
SgFr_dep	1
SgFr_new_answers	FALSE

p(3,Z)	
▪ SgFr_state	LOOPING_EVAL
SgFr_dfn	3
SgFr_dep	1
SgFr_new_answers	FALSE

p(4,Z)	
▪ SgFr_state	LOOPING_EVAL
SgFr_dfn	4
SgFr_dep	1
▪ SgFr_new_answers	TRUE

Figure 5.14: Detecting several partial fix-points not corresponding to completion

`p(3,3)`, when consuming the answer `p(2,3)` after step 20. Thus, when returning to `p(3,Z)`, we re-execute one more time its looping alternative (step 24 in Fig. 5.14) but no new answers are found. Next, subgoal `p(3,Z)` is re-marked as `LOOPING` (step 29 in Fig. 5.14), the execution fails to `p(2,Z)`, and because no new answers were found for `p(2,Z)` it is also marked as `LOOPING` (step 30 in Fig. 5.14).

Again, the same happens when the computation returns to `p(1,Z)`, we also find four answers for it: `p(1,1)`, `p(1,2)`, `p(1,4)` and `p(1,3)`, which will force the re-execution of its looping alternative (step 31 in Fig. 5.14). Subgoals `p(2,Z)`, `p(3,Z)`, `p(4,Z)` and `p(1,Z)` are then called again and, in the continuation, we find three new answers for `p(4,Z)`: `p(4,2)`, `p(4,4)` and `p(4,3)` when consuming answers from `p(1,Z)` after step 36. When returning to `p(4,Z)`, we re-execute a last time its looping alternative (step 37 in Fig. 5.14) but no more answers are found. We then backtrack up to `p(1,Z)` and all subgoals are marked as `COMPLETE` (step 42 in Fig. 5.14).

Chapter 6

SLDT Linear Tabling

This chapter describes the design and implementation of the second linear tabling mechanism that we have implemented, the SLDT linear tabling mechanism as originally proposed by Zhou *et al.* [ZSYY00]. As for the previous mechanisms, it follows a scheduling strategy based on the local scheduling strategy.

We start by describing the basic execution model for SLDT and by presenting an example showing how a tabled program is transformed to include specific tabling primitives for this mechanism. We then present an evaluation example and provide the details for implementing SLDT evaluation as an external Prolog module in Yap.

6.1 Basic Execution Model

In the SLDT execution model, each tabled call can be a generator and a consumer as well. A first call to a tabled subgoal is called a *pioneer* and repeated calls to tabled subgoals are called *followers* of the pioneer. For every pioneer and its followers there is a single table entry associated with them.

The basic idea is as follows. The SLDT strategy constructs an SLD tree in the same left-to-right and depth-first fashion as the SLD resolution except when a tabled subgoal call is a follower. In this case, we first use the answers available from its table entry to resolve the call. When no more unconsumed answers are available, we resolve the call by using the remaining clauses of the latest former call. Using the terminology in [ZSYY00], we say that the current call *steals* the choice point from the latest former call.

When backtracking, to a pioneer or a follower, we use the same strategy, we first try to consume answers and then we try to use the clauses. After we exhaust all the answers and clauses, we simply fail if not a pioneer. Otherwise, we should decide whether it is necessary to re-execute the pioneer starting from the first clause of the tabled predicate. Re-execution will be repeated until no new answers can be generated, that is, until reaching a fix-point.

For computations with multiple calls containing recursive calls, followers may be encountered several times. Each follower executes from the backtracking point of the former repeated call, that is, each follower skips the previous clauses leading to repeated calls. The answers that may potentially be lost by skipping these clauses are found by re-computation. However, in the case of multiple calls with repeated calls, re-computation will be quite complicated because new answers can be possibly generated by exploring the multiple calls in different combinations. Thus, the whole computation has to be re-explored one or more times from the first tabled call to guarantee that all answers have been obtained and none missed.

6.2 Program Transformation

Figure 6.1 uses again the right recursive `path/2` program from Fig. 2.3 to illustrate how programs are transformed and augmented with tabling primitives to implement SLDT evaluation.

```
path(X,Z):- tabled_call(path(X,Z),Sid,Cid),
            sldt_loop(Sid,Cid,Alt),
            (Alt=0 -> consume_answer(path(X,Z),Sid,Cid)
                    ; path0(path(X,Z),Sid,Alt)).

path0(path(X,Z),Sid,1):- edge(X,Y), path(Y,Z), new_answer(path(X,Z),Sid).
path0(path(X,Z),Sid,2):- edge(X,Z), new_answer(path(X,Z),Sid).
path0(path(X,Z),Sid,3):- fixpoint_check(Sid).
```

Figure 6.1: Program transformation for the right recursive `path/2` program

As for the previous mechanisms, the `path/2` clause is maintained so that we can call it from other predicates without any change. The tabling primitive `tabled_call/3` in the body of `path/2` inserts/checks for the subgoal in the table space and returns a *subgoal id* (`Sid`) and a *consumer id* (`Cid`). The subgoal id represents the subgoal frame in the table space and the consumer id represents a *consumer frame* used to keep track of the last consumed answer for each call. A pioneer and its followers share the same subgoal id, but each one has a different consumer id.

The `sldt_loop/3` primitive controls the evaluation process and the `consume_answer/3` primitive implements the process of consuming answers one at a time. Both primitives are implemented as backtrackable predicates. The `Alt` argument in the `sldt_loop/3` primitive is used to switch between consuming answers (cases where `Alt` is bound to zero) or explore clauses (cases where `Alt` is bound to the number of the clause to be explored).

Each clause in the original definition of `path/2` becomes a clause for a new distinct predicate, `path0/3` in the example, with 3 arguments. The first argument is the previous head clause; the second is the subgoal id; and the third is the number of the clause regarding the textual order it appears on the program. A `new_answer/2` primitive is added to the end of each clause body. In addition, an extra clause is used to check for fix-points after we exhaust all the answers and clauses.

6.3 An Evaluation Example

Figures 6.2 to 6.5 show the evaluation sequence for the query goal `p(1,Z)` if applying the program transformation for SLDT evaluation presented in Fig. 6.1. At the top, each figure illustrates the program code and the state of the table space at the end of the evaluation represented in the figure. The bottom sub-figures show the resulting forest of trees with the numbering of nodes denoting the evaluation sequence. For illustration purposes the program code was simplified, predicates `consume_answer/3`, `path/2`, `path0/3` and `edge/2` are respectively denoted as `consume/3`, `p/2`, `p0/3` and `e/2`.

The evaluation begins with the `tabled_call/3` primitive being called for the `p(1,Z)` subgoal. As `p(1,Z)` is a pioneer (pioneers are represented by black oval boxes), a new subgoal frame, with id `sid1`, and a new consumer frame, with id `cid1`, are allocated in the table space for it (step 2). The `sldt_loop/3` primitive then starts exploring the first alternative of `p0/3` (`Alt` is bound to 1) and, in the continuation, `p(2,Z)` is first called (step 6).

As `p(2,Z)` is a pioneer, we add a new subgoal frame, with id `sid2`, and a new consumer frame, with id `cid2` (step 7), and proceed with `p(2,Z)` being resolved against the first clause for `p0/3` (step 9). In the continuation, subgoal `p(1,Z)` is called again (step 11).

```
p(X,Z):- tabled_call(p(X,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt),
         (Alt=0 -> consume(p(X,Z),Sid,Cid) ; p0(p(X,Z),Sid,Alt)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,3):- fixpoint_check(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Cid	
sid1	2. p(1,Z)	16. p(1,2)	2. cid1	
			12. cid3	19. p(1,2)
sid2	7. p(2,Z)	20. p(2,2)	7. cid2	

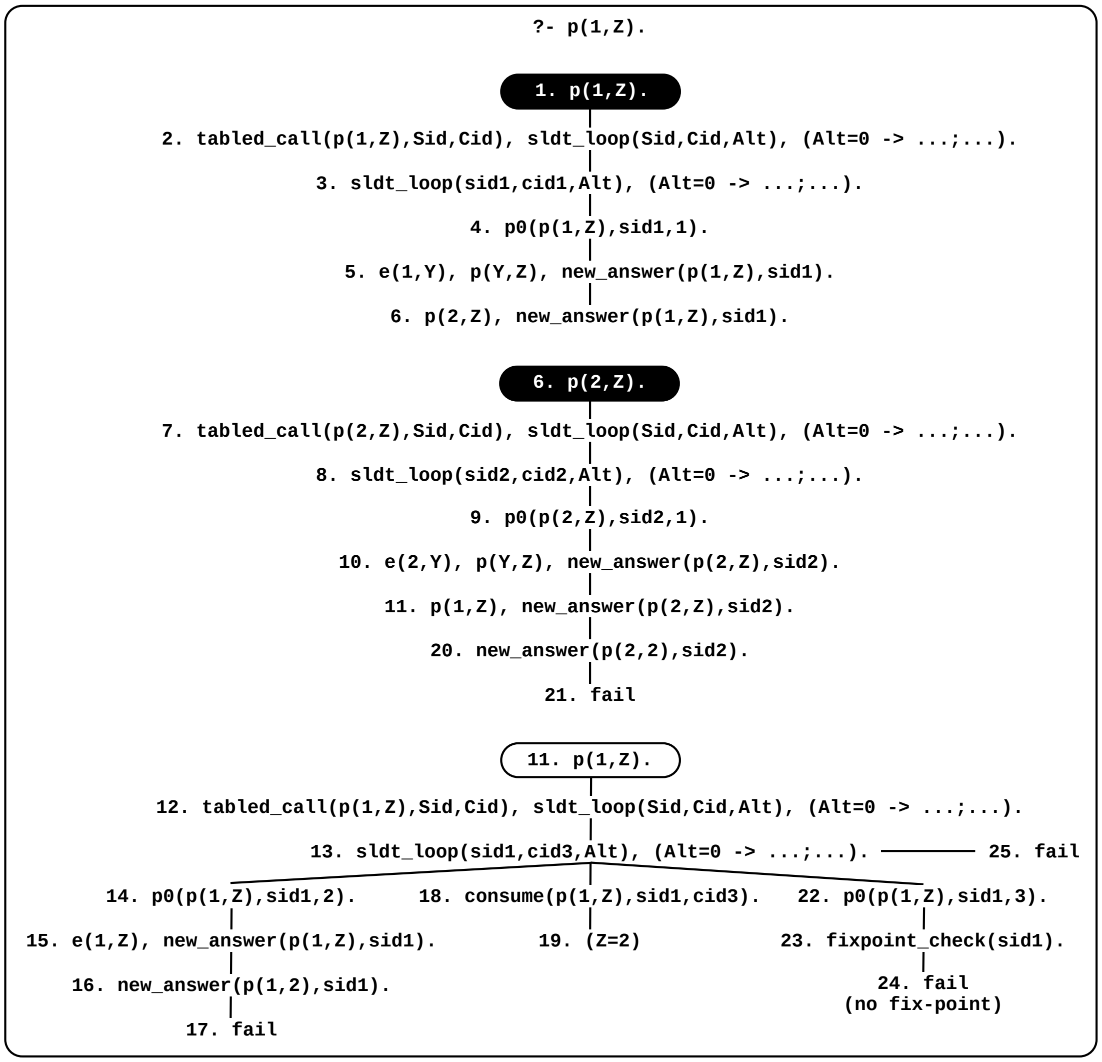

?- p(1,Z).
1. p(1,Z).
2. tabled_call(p(1,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt), (Alt=0 -> ...;...).
3. sldt_loop(sid1,cid1,Alt), (Alt=0 -> ...;...).
4. p0(p(1,Z),sid1,1).
5. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1).
6. p(2,Z), new_answer(p(1,Z),sid1).
6. p(2,Z).
7. tabled_call(p(2,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt), (Alt=0 -> ...;...).
8. sldt_loop(sid2,cid2,Alt), (Alt=0 -> ...;...).
9. p0(p(2,Z),sid2,1).
10. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2).
11. p(1,Z), new_answer(p(2,Z),sid2).
20. new_answer(p(2,2),sid2).
21. fail
11. p(1,Z).
12. tabled_call(p(1,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt), (Alt=0 -> ...;...).
13. sldt_loop(sid1,cid3,Alt), (Alt=0 -> ...;...).
25. fail
14. p0(p(1,Z),sid1,2).
18. consume(p(1,Z),sid1,cid3).
22. p0(p(1,Z),sid1,3).
15. e(1,Z), new_answer(p(1,Z),sid1).
19. (Z=2)
23. fixpoint_check(sid1).
16. new_answer(p(1,2),sid1).
24. fail
(no fix-point)
17. fail

Figure 6.2: SLDT evaluation for the right recursive `path/2` program (steps 1 to 25)

Since `p(1,Z)` is now a follower (followers are represented by white oval boxes) and no answers are still available for it, the `sldt_loop/3` primitive tries the next unexplored alternative of `p0/3` for `p(1,Z)`, alternative two in this case (step 14).

In the continuation, we obtain a first answer for `p(1,Z)` (step 16). The answer is inserted in the table and, because we are following a local scheduling strategy, the execution fails. We then backtrack to the `sldt_loop/3` primitive and now we can consume answers (step 18). The answer `p(1,2)` is thus consumed and the consumer frame for the current follower, `cid3`, is made to point to it (step 19). The binding `Z=2` is then propagated to the context of subgoal `p(2,Z)` and a first answer for `p(2,Z)` is also found (step 20). The evaluation then fails and we backtrack again to node 13. The `sldt_loop/3` primitive calls the third alternative of `p0/3` for `p(1,Z)` (step 22) and we check for a fix-point (step 23). We have found a new answer for `p(1,Z)`, so we declare the subgoal as *no fix-point*, meaning that we need to re-execute it starting from its first clause, and fail (step 24). Note that to implement local scheduling we also need to fail when executing the `fixpoint_check/1` primitive. The evaluation then backtracks to node 13 and, because the current subgoal call for `p(1,Z)` is not a pioneer, we also fail (step 25). Re-execution should be handled by the pioneer, as otherwise we may lose part of the computation (please see subsection 6.4.3 for more details).

The evaluation then backtracks to the `sldt_loop/3` primitive at node 8, and we consume the answer `p(2,2)` (step 27). The binding `Z=2` is then propagated to the context of subgoal `p(1,Z)`, and a new answer for `p(1,Z)` is found (step 28). However, this last answer repeats the answer found in step 16, so we fail and backtrack again to node 8. We then explore the second alternative of `p0/3` for `p(2,Z)` (step 30) obtaining, in the continuation, a second answer for `p(2,Z)` (step 32) and a second answer for `p(1,Z)` (step 36). The evaluation then proceeds with the third alternative of `p0/3` for `p(2,Z)` (step 38) and we check for a fix-point (step 39). We have found two new answers for `p(2,Z)`, so we declare the subgoal as *no fix-point* and fail (step 40).

We then backtrack one more time to node 8 and, because the current subgoal call for `p(2,Z)` is a pioneer, we re-execute the subgoal starting from its first alternative (step 41). The execution then proceeds with another follower being called for subgoal `p(1,Z)` (step 43). The `sldt_loop/3` primitive for this new follower (step 45) starts by consuming the available answers for `p(1,Z)`. The bindings `Z=2` (step 47) and `Z=1` (step 50) are propagated to the context of subgoal `p(2,Z)`, but only redundant answers are found (steps 48 and 51). We return to the `sldt_loop/3` primitive for the follower (node 45) and then we try to use the clauses. Since we have failed to detect a fix-point in step 24, we need to re-start the computation of `path(1,Z)` from the beginning, as

```
p(X,Z):- tabled_call(p(X,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt),
         (Alt=0 -> consume(p(X,Z),Sid,Cid) ; p0(p(X,Z),Sid,Alt)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,3):- fixpoint_check(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Cid	
sid1	2. p(1,Z)	16. p(1,2) 36. p(1,1)		
sid2	7. p(2,Z)	20. p(2,2) 32. p(2,1)	7. cid2	27. p(2,2) 35. p(2,1)

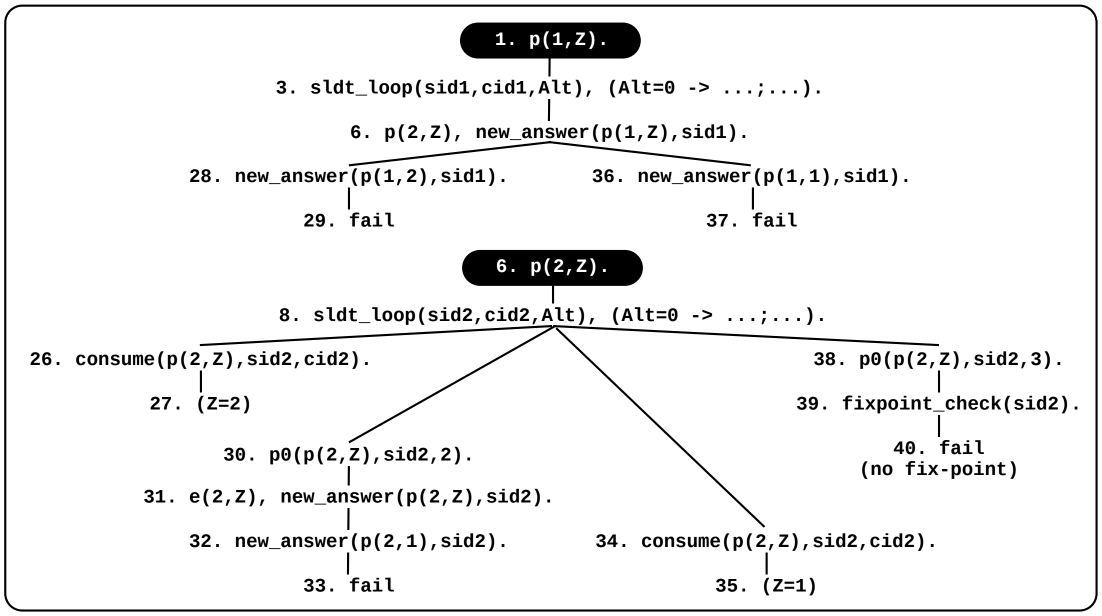

1. p(1,Z).
3. sldt_loop(sid1,cid1,Alt), (Alt=0 -> ...;...).
6. p(2,Z), new_answer(p(1,Z),sid1).
28. new_answer(p(1,2),sid1).
29. fail
36. new_answer(p(1,1),sid1).
37. fail
6. p(2,Z).
8. sldt_loop(sid2,cid2,Alt), (Alt=0 -> ...;...).
26. consume(p(2,Z),sid2,cid2).
27. (Z=2)
30. p0(p(2,Z),sid2,2).
31. e(2,Z), new_answer(p(2,Z),sid2).
32. new_answer(p(2,1),sid2).
33. fail
34. consume(p(2,Z),sid2,cid2).
35. (Z=1)
38. p0(p(2,Z),sid2,3).
39. fixpoint_check(sid2).
40. fail
(no fix-point)

Figure 6.3: SLDT evaluation for the right recursive `path/2` program (steps 26 to 40)

otherwise we may lose answers for the other subgoals in evaluation.

The evaluation of the first clause of `p0/3` for `p(1,Z)` at step 53 leads, in the continuation, to another follower, but this time to subgoal `p(2,Z)` (step 55). Again, the `sldt_loop/3` primitive for this new follower (step 57) starts by consuming the available answers for it (step 58), but no new answers are found. When backtracking, the `sldt_loop/3` primitive at node 57 executes the unexplored alternatives of `p0/3` for `p(2,Z)`. First it tries alternative two, but no new answers are found (steps 65 to 68), and then it checks for a fix-point using alternative three (step 69). Since we have found no new answers during the last re-execution of subgoal `p(2,Z)`, we declare the subgoal as *partial fix-point*, meaning that we may have reached a fix-point, and fail (step 71). The evaluation then backtracks to node 57 and, because the current call

```
p(X,Z):- tabled_call(p(X,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt),
         (Alt=0 -> consume(p(X,Z),Sid,Cid) ; p0(p(X,Z),Sid,Alt)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,3):- fixpoint_check(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Cid	
sid1	2. p(1,Z)	16. p(1,2) 36. p(1,1)	44. cid4	47. p(2,2) 50. p(2,1)
sid2	7. p(2,Z)	20. p(2,2) 32. p(2,1)	56. cid5	59. p(2,2) 62. p(2,1)

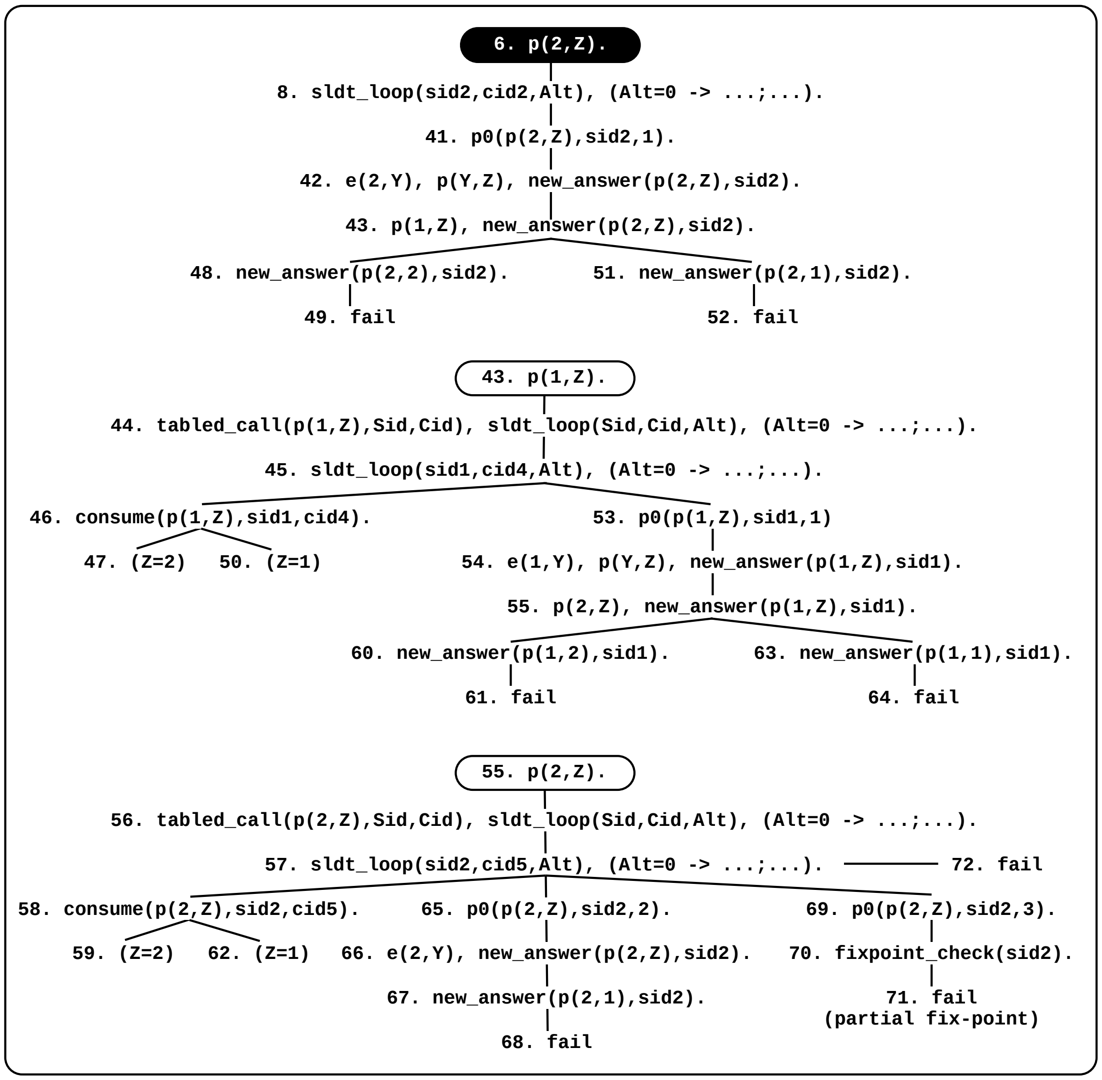

6. p(2,Z).
8. sldt_loop(sid2,cid2,Alt), (Alt=0 -> ...;...).
41. p0(p(2,Z),sid2,1).
42. e(2,Y), p(Y,Z), new_answer(p(2,Z),sid2).
43. p(1,Z), new_answer(p(2,Z),sid2).
48. new_answer(p(2,2),sid2).
51. new_answer(p(2,1),sid2).
49. fail
52. fail
43. p(1,Z).
44. tabled_call(p(1,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt), (Alt=0 -> ...;...).
45. sldt_loop(sid1,cid4,Alt), (Alt=0 -> ...;...).
46. consume(p(1,Z),sid1,cid4).
53. p0(p(1,Z),sid1,1)
47. (Z=2)
50. (Z=1)
54. e(1,Y), p(Y,Z), new_answer(p(1,Z),sid1).
55. p(2,Z), new_answer(p(1,Z),sid1).
60. new_answer(p(1,2),sid1).
63. new_answer(p(1,1),sid1).
61. fail
64. fail
55. p(2,Z).
56. tabled_call(p(2,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt), (Alt=0 -> ...;...).
57. sldt_loop(sid2,cid5,Alt), (Alt=0 -> ...;...).
72. fail
58. consume(p(2,Z),sid2,cid5).
65. p0(p(2,Z),sid2,2).
69. p0(p(2,Z),sid2,3).
59. (Z=2)
62. (Z=1)
66. e(2,Y), new_answer(p(2,Z),sid2).
70. fixpoint_check(sid2).
67. new_answer(p(2,1),sid2).
71. fail
(partial fix-point)
68. fail

Figure 6.4: SLDT evaluation for the right recursive `path/2` program (steps 41 to 72)

is not a pioneer, we also fail (step 72). Note that because followers avoid trying the same alternatives as the former call, the computation may be incomplete, and thus we should fail in order to fully exploit the current branch up to the pioneer. Therefore, declaring a subgoal as *complete* is only safe when reaching the pioneer, as otherwise we may lose part of the computation.

```
p(X,Z):- tabled_call(p(X,Z),Sid,Cid), sldt_loop(Sid,Cid,Alt),
         (Alt=0 -> consume(p(X,Z),Sid,Cid) ; p0(p(X,Z),Sid,Alt)).

p0(p(X,Z),Sid,1):- e(X,Y), p(Y,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,2):- e(X,Z), new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid,3):- fixpoint_check(Sid).

e(1,2).
e(2,1).
```

Sid	Subgoal	Answers	Cid	
sid1	2. p(1,Z)	16. p(1,2) 36. p(1,1) 83. complete	2. cid1	85. p(2,2) 87. p(2,1)
sid2	7. p(2,Z)	20. p(2,2) 32. p(2,1) 81.complete		

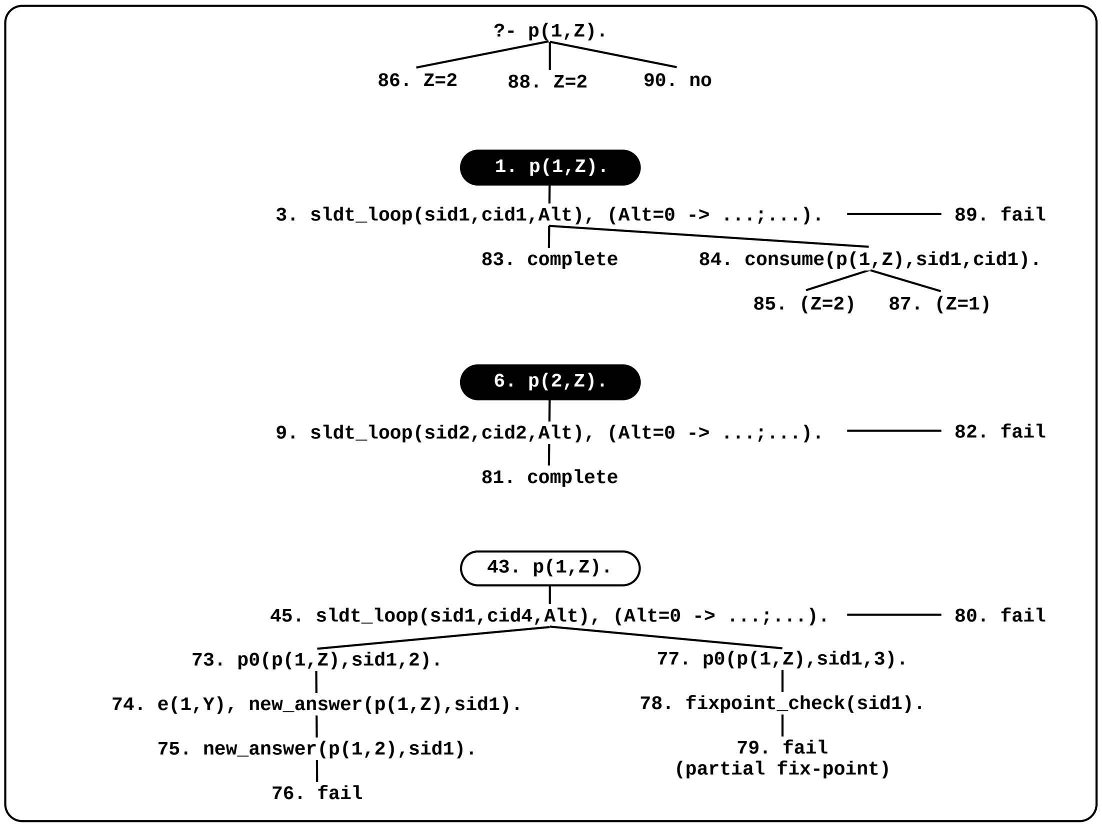

?- p(1,Z).
86. Z=2
88. Z=2
90. no
1. p(1,Z).
3. sldt_loop(sid1,cid1,Alt), (Alt=0 -> ...;...).
89. fail
83. complete
84. consume(p(1,Z),sid1,cid1).
85. (Z=2)
87. (Z=1)
6. p(2,Z).
9. sldt_loop(sid2,cid2,Alt), (Alt=0 -> ...;...).
82. fail
81. complete
43. p(1,Z).
45. sldt_loop(sid1,cid4,Alt), (Alt=0 -> ...;...).
80. fail
73. p0(p(1,Z),sid1,2).
74. e(1,Y), new_answer(p(1,Z),sid1).
75. new_answer(p(1,2),sid1).
76. fail
77. p0(p(1,Z),sid1,3).
78. fixpoint_check(sid1).
79. fail
(partial fix-point)

Figure 6.5: SLDT evaluation for the right recursive `path/2` program (steps 73 to 90)

The same evaluation sequence occurs when we backtrack to the `sldt_loop/3` primitive at node 45 and we execute the unexplored alternatives of `p0/3` for `p(1,Z)`. No new answers are found when exploring alternative two (steps 73 to 76), and we declare the subgoal as *partial fix-point* when executing alternative three (steps 77 to 79). Since the call for the `sldt_loop/3` primitive at node 45 is a follower, we also fail (step 80).

Backtracking sends us back to the `sldt_loop/3` primitive for the pioneer of `p(2,Z)` (node 9). As `p(2,Z)` is declared as *partial fix-point* and no new answers were found since then, we can declare `p(2,Z)` to be *complete* (step 81) and fail (step 82). Note that we can fail because we have already consumed the two available answers for this pioneer at steps 27 and 35, using the consumer frame `cid2`.

A similar situation occurs when we backtrack to the `sldt_loop/3` primitive for the pioneer of `p(1,Z)` (node 3). Subgoal `p(1,Z)` is also declared as *partial fix-point* and no new answers were found since then, and thus we can declare `p(1,Z)` to be *complete* (step 83). Before failing at step 89, we first consume the two available answers for `p(1,Z)` at steps 85 and 87, using the consumer frame `cid1`, obtaining, in the continuation, the two answers for the query goal (steps 86 and 88). Finally, at step 90, we return *no* to the query goal.

6.4 Implementation Details

We next give a detailed description of all the implementation issues required to fully support SLDT evaluation.

6.4.1 Subgoal Frames

In the SLDT linear tabling mechanism, a subgoal frame is a seven field data structure. Four of the these fields are the usual `SgFr_state`, `SgFr_answers`, `SgFr_first_answer` and `SgFr_last_answer` fields as implemented in the previous tabling mechanisms. The three extra fields have the following meaning:

- `SgFr_next_alt:` is the number of the next clause to be tried.
- `SgFr_new_answers:` indicates if new answers were found since the last time we (re-)started the computation from the first clause.
- `SgFr_pioneer:` is a pointer to the consumer frame of the pioneer. It is used to detect whether a call is a pioneer or a follower.

The corresponding pseudo-code for the `new_subgoal_frame()` procedure is presented next in Fig. 6.6. Figure 6.7 uses the example from Figures 6.2 to 6.5 to illustrate how these fields are used and updated during a tabled evaluation.

```
new_subgoal_frame() {
  sf = allocate_subgoal_frame();
  SgFr_state(sf) = READY;
  SgFr_answers(sf) = open_trie();
  SgFr_first_answer(sf) = NULL;
  SgFr_last_answer(sf) = NULL;
  SgFr_next_alt(sf) = 1;
  SgFr_new_answers(sf) = FALSE;
  SgFr_pioneer(sf) = NULL;
  return sf;
}
```

Figure 6.6: Pseudo-code for `new_subgoal_frame()`

Figure 6.7(a) shows the state of the table space after calling the `tabled_call/3` primitive for the `p(1,Z)` subgoal (step 2 in Fig. 6.2). As `p(1,Z)` is a pioneer, the `tabled_call/3` primitive allocates and initializes a new subgoal frame in the table space for it. The `sldt_loop/3` primitive then starts exploring the first alternative of `p0/3` for `p(1,Z)` and the `SgFr_next_alt` field of the subgoal frame for `p(1,Z)` is updated to 2, the next alternative to be taken. In the continuation, the `tabled_call/3` primitive is called for the `p(2,Z)` subgoal and a new subgoal frame is also allocated and initialized (step 7 in Fig. 6.2). Figure 6.7(b) shows the resulting state of both subgoal frames at that point.

Subgoal `p(1,Z)` is then called again (step 11 in Fig. 6.2) and the `sldt_loop/3` primitive executes the alternative stored in the `SgFr_next_alt` field of the subgoal frame for `p(1,Z)`, alternative two in this case, and updates the `SgFr_next_alt` field to 3 (step 14 in Fig. 6.2). The execution then continues and we obtain a first answer for `p(1,Z)` (step 16 in Fig. 6.2). Figure 6.7(c) shows the resulting state of the subgoal frame for `p(1,Z)` at that point.

The evaluation then obtains a first answer for `p(2,Z)` (step 20 in Fig. 6.2) and second answers for `p(2,Z)` and `p(1,Z)` (steps 32 and 36 in Fig. 6.3). Later, subgoal `p(2,Z)` is first declared as *partial fix-point* (step 71 in Fig. 6.4) and then subgoal `p(1,Z)` is also declared as *partial fix-point* (step 79 in Fig. 6.5). Backtracking then sends us back to the `sldt_loop/3` primitive for the pioneer of `p(2,Z)` where it is declared as *complete* (step 81 in Fig. 6.5). Figure 6.7(d) shows the resulting state of both subgoal frames at that point (a subgoal declared as *partial fix-point* is marked with the flag `P_FIXPOINT` in the `SgFr_state` field). Finally, a similar situation occurs when we backtrack to the

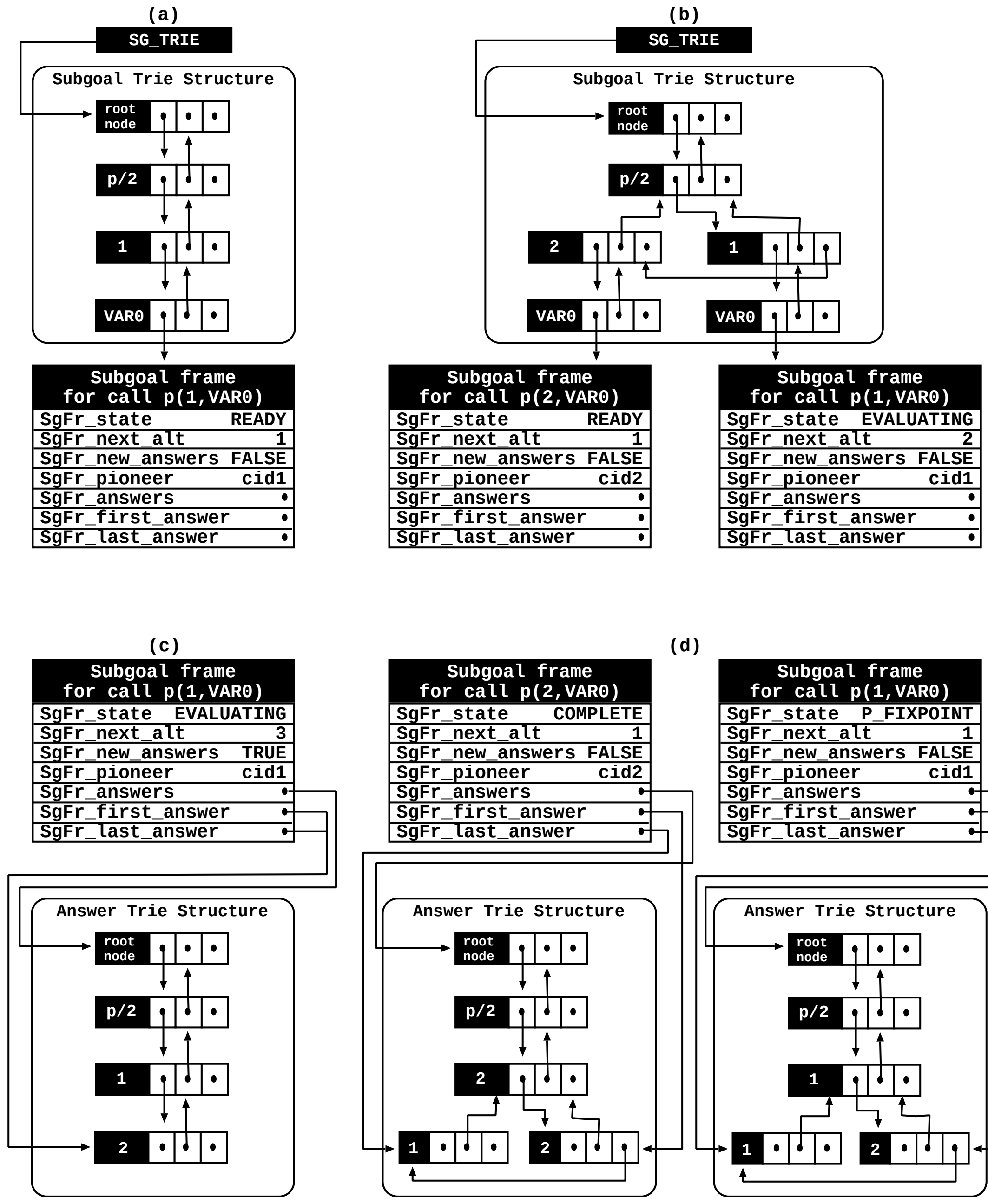


Figure 6.7: Subgoal frames at different points of the evaluation of the right recursive `path/2` program of Figures 6.2 to 6.5

`sldt_loop/3` primitive for the pioneer of `p(1,Z)` where it is also declared as *complete* (step 83 in Fig. 6.5).

6.4.2 Tabling Primitives

We next show the implementation details for the five tabling primitives that support SLDT evaluation. We start with Fig. 6.8 showing the pseudo-code for the `new_answer/2` primitive.

```
new_answer(YAP_Term ANSWER, SgFr SF) {
  if (put_answer(SF, ANSWER) == TRUE)
    SgFr_new_answers(SF) = TRUE;
  return FALSE;                                     // always fail at the end
}
```

Figure 6.8: Pseudo-code for the `new_answer/2` primitive

The `new_answer/2` primitive calls the `put_answer()` procedure to insert the given `ANSWER` in the answer trie structure for the `SF` subgoal frame and, if the answer is new, it updates the `SgFr_new_answers` to `TRUE`. We then implement a local scheduling approach and always fail at the end.

When we reach the last clause of a tabled predicate, we execute the `fixpoint_check/1` primitive. Figure 6.9 shows the pseudo-code for it.

```
fixpoint_check(SgFr SF) {
  if (SgFr_new_answers(SF) == TRUE)            // check for a partial fix-point
    SgFr_state(SF) = NO_FIXPOINT;                                // no fix-point
  else
    SgFr_state(SF) = P_FIXPOINT;                            // partial fix-point
  SgFr_next_alt(SF) = 1;                                 // reset the clause counter
  return FALSE;                                         // always fail at the end
}
```

Figure 6.9: Pseudo-code for the `fixpoint_check/1` primitive

The `fixpoint_check/1` primitive starts by checking if new answers were found during the last traversal of the predicate clauses. If so (cases where `SgFr_new_answers` is `TRUE`), it changes the subgoal's state to `NO_FIXPOINT`. Otherwise, it updates the subgoal's state to `P_FIXPOINT`. In both cases it resets the `SgFr_next_alt` field to 1 and fails. The computation then returns to the `sldt_loop/3` primitive for the tabled subgoal call in execution, and the `NO_FIXPOINT` and `P_FIXPOINT` states are then used to decide whether it should fail, re-execute the subgoal from its first clause or complete.

We next describe the 3 primitives that implement the execution control for the entry clause of each tabled predicate. We first show in Fig. 6.10 the pseudo-code for the `tabled_call/3` primitive.

```
tabled_call(YAP_Term SUBGOAL_CALL, SgFr SF, ConsFr CF) {
  sf = put_tabled_call(SUBGOAL_CALL);
  cf = add_consumer_frame(sf);
  ConsFr_LastAnswer(cf) = NULL;
  if (SgFr_state(sf) == READY) {                         // new subgoal call
    SgFr_state(sf) = EVALUATING;
    SgFr_pioneer(sf) = cf;
  }
  YAP_Unify(SF, sf);
  YAP_Unify(CF, af);
  return TRUE;                                   // always succeed at the end
}
```

Figure 6.10: Pseudo-code for the `tabled_call/3` primitive

As for the previous mechanisms, the `tabled_call/3` primitive starts by calling the `put_tabled_call()` procedure in order to insert the given `SUBGOAL_CALL` in the subgoal trie structure. Next, it adds a new consumer frame to the obtained subgoal frame and initializes it with `NULL`, meaning that no answers were still consumed for the current call. Then, if the resulting subgoal frame is new, it changes the subgoal's state to `EVALUATING` and updates the `SgFr_pioneer` field to the current consumer frame. Finally, the `SF` and `CF` arguments are unified with the corresponding pointers to the subgoal and consumer frames and the procedure succeeds.

The `sldt_loop/3` primitive then controls the evaluation process. It was implemented in Yap as a backtrackable predicate. This is a special case of a backtrackable predicate because it uses the same procedure, the `sldt_loop()` procedure, to start the execution of the primitive and to continue its execution when backtracking occurs. Figure 6.11 shows its implementation. Initially, the procedure checks if the computation is returning from the `fixpoint_check/1` primitive with the subgoal's state as `NO_FIXPOINT`. If so, it changes the subgoal's state to `EVALUATING` and, if is not a pioneer, fails. Otherwise, it checks if the subgoal's state is `P_FIXPOINT` and, if new answers were found in the meantime (cases where the `SgFr_new_answers` field is `TRUE`), it changes the subgoal's state to `EVALUATING` in order to re-execute the subgoal from the beginning. Otherwise, if the current subgoal is a pioneer, then we know that no new answers were found during the last time we (re-)executed the subgoal and thus we can safely mark it as `COMPLETE`.

In the continuation, we first try to consume answers, cases where we succeed with the `NEXT_ALT` argument bound to zero, and if no unconsumed answers are available then we try to use the clauses, cases where we succeed with the `NEXT_ALT` argument bound to the number of the next clause to be explored. When the subgoal is `COMPLETE` or

```
sldt_loop(SgFr SF, CallFr CF, YAP_Int NEXT_ALT) {
  if (SgFr_state(sf) == NO_FIXPOINT) {
    SgFr_state(SF) = EVALUATING;
    if (SgFr_pioneer(sf) != CF) {
      delete_consumer_frame(CF);
      YAP_cut_fail();
      return FALSE;
    }
  } else if (SgFr_state(sf) == P_FIXPOINT) {
    if (SgFr_new_answers(sf) == TRUE)
      SgFr_state(SF) = EVALUATING;
    else if (SgFr_pioneer(sf) == CF)
      SgFr_state(SF) = COMPLETE;
  }
  if (CallFr_LastAnswer(CF) != SgFr_last_answer(SF)) {
    YAP_Unify(NEXT_ALT, 0);                               // consume answers
    return TRUE;
  }
  if (SgFr_state(SF) == COMPLETE || SgFr_state(sf) == P_FIXPOINT) {
    delete_consumer_frame(CF);
    YAP_cut_fail();
    return FALSE;
  }
  if (SgFr_next_alt(sf) == 1)    // before (re-)starting the computation ...
    SgFr_new_answers(SF) = FALSE;         // ... reset the new answers field
  YAP_Unify(NEXT_ALT, SgFr_next_alt(SF));             // try the next clause
  SgFr_next_alt(SF)++;
  return TRUE;
}
```

Figure 6.11: Pseudo-code for the `sldt_loop/3` primitive

`P_FIXPOINT` we avoid trying the clauses and simply fail.

We end with Fig. 6.12 showing the pseudo-code for the `consume_answer/3` primitive. The `consume_answer/3` primitive was implemented in Yap as a backtrackable predicate. The `consume_answer_init()` procedure is executed when the primitive is first called, and the `consume_answer_cont()` procedure is executed when backtracking occurs. The consumer frame pointer, argument `CF`, is used to keep track of the last returned answer when backtracking.

6.4.3 Detecting Completion

In the SLDT execution model, completion is detected in two steps. First, we use the `fixpoint_check/1` primitive to detect when no new answers were found during the last traversal of the predicate clauses and, for such cases, we mark the subgoal as

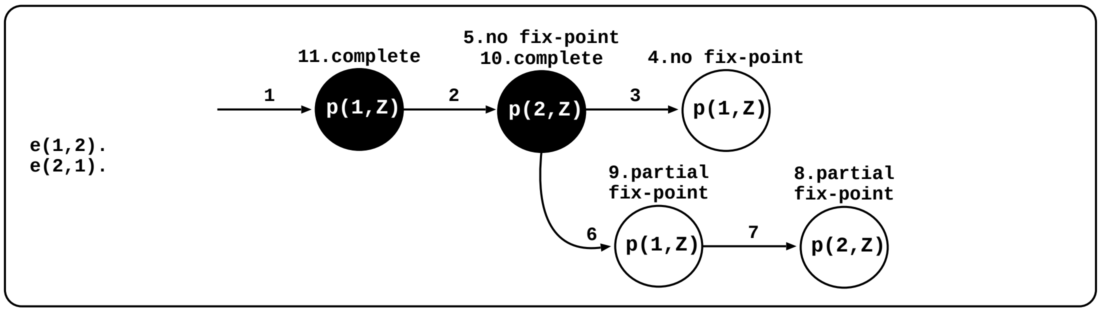
```
consume_answer_init(YAP_Term SUBGOAL_CALL, SgFr SF, ConsFr CF) {
  leaf = ConsFr_LastAnswer(CF);
  if (ans == NULL)                                // get first answer leaf node
    leaf = SgFr_first_answer(SF);
  else                                             // get next answer leaf node
    leaf = TrNode_child(ans);
  answer = get_trie_entry(leaf);                      // load answer from trie
  YAP_Unify(SUBGOAL_CALL, answer);
  ConsFr_LastAnswer(CF) = leaf;                     // update for backtracking
  return TRUE;
}

consume_answer_cont(YAP_Term SUBGOAL_CALL, SgFr SF, ConsFr CF) {
  leaf = ConsFr_LastAnswer(CF);
  if (leaf == SgFr_last_answer(SF)) {                        // no more answers
    YAP_cut_fail();
    return FALSE;
  } else                                           // get next answer leaf node
    leaf = TrNode_child(ans);
  answer = get_trie_entry(leaf);                      // load answer from trie
  YAP_Unify(SUBGOAL_CALL, answer);
  ConsFr_LastAnswer(CF) = leaf;                     // update for backtracking
  return TRUE;
}
```

Figure 6.12: Pseudo-code for the `consume_answer/3` primitive

`P_FIXPOINT`. Second, a subgoal marked as `P_FIXPOINT` is considered to be completely evaluated when the computation returns to the pioneer and no new answers were found since the last execution of the `fixpoint_check/1` primitive. Figure 6.13 uses again the example from Figures 6.2 to 6.5 to illustrate how the subgoal fields `SgFr_state` and `SgFr_new_answers` are used to detect completion. The black dots in the sub-figure below indicates the fields being updated at each step of the example.

Initially, `p(1,Z)` and `p(2,Z)` are first called and two subgoal frames are allocated. Next, `p(2,Z)` calls a follower of `p(1,Z)` and we obtain first answers for `p(1,Z)` and `p(2,Z)`. When executing the `fixpoint_check/1` primitive for the follower, `p(1,Z)` is thus marked as `NO_FIXPOINT` (step 4 in Fig. 6.13). In the continuation, we obtain second answers for `p(2,Z)` and `p(1,Z)` and when executing the `fixpoint_check/1` primitive for `p(2,Z)`, it is also marked as `NO_FIXPOINT` (step 5 in Fig. 6.13).

The execution then proceeds with new followers for `p(1,Z)` and `p(2,Z)`. Since we have failed to detect a fix-point during the last execution of the `fixpoint_check/1` primitive for both subgoals, we re-start the computation from the beginning for each follower and reset the `SgFr_new_answers` field to `FALSE`. Later, subgoal `p(2,Z)` is first marked as `P_FIXPOINT` (step 8 in Fig. 6.13) and then subgoal `p(1,Z)` is also marked



(4)

	p(1,Z)	
	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE

	p(2,Z)	
	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE

(5)

	p(1,Z)	
▪	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE

	p(2,Z)	
▪	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE

(8)

	p(1,Z)	
	SgFr_state	EVALUATING
▪	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	P_FIXPOINT
▪	SgFr_new_answers	FALSE

(9)

	p(1,Z)	
▪	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE

	p(2,Z)	
	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE

(10)

	p(1,Z)	
	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE

	p(2,Z)	
▪	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(11)

	p(1,Z)	
▪	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

	p(2,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

Figure 6.13: Detecting completion when evaluating the right recursive `path/2` program of Figures 6.2 to 6.5

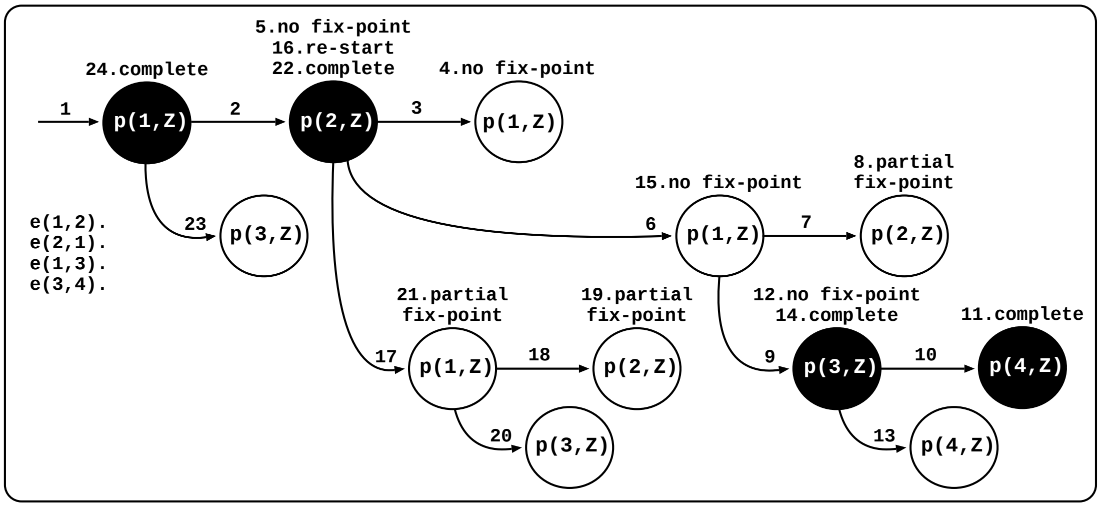
as `P_FIXPOINT` (step 9 in Fig. 6.13).

Backtracking then sends us back to the pioneer of `p(2,Z)` where it is marked as `COMPLETE` (step 10 in Fig. 6.13). Finally, when backtracking to the pioneer of `p(1,Z)` it is also marked as `COMPLETE` (step 11 in Fig. 6.13).

Figure 6.14 presents a more complex example that illustrates better how the subgoal fields `SgFr_state` and `SgFr_new_answers` are used to detect completion.

Initially, `p(1,Z)` and `p(2,Z)` are first called and two subgoal frames are allocated. Next, `p(2,Z)` calls a follower of `p(1,Z)` and we obtain two answers for each subgoal: `p(1,2)` and `p(1,3)` for subgoal `p(1,Z)` and `p(2,2)` and `p(2,3)` for subgoal `p(2,Z)`. Next, the `fixpoint_check/1` primitive for the follower marks `p(1,Z)` as `NO_FIXPOINT` (step 4 in Fig. 6.14). In the continuation, we obtain a third answer for `p(2,Z)`, `p(2,1)`, and a third answer for `p(1,Z)`, `p(1,1)`. Then, the `fixpoint_check/1` primitive for `p(2,Z)` also marks `p(2,Z)` as `NO_FIXPOINT` (step 5 in Fig. 6.14).

The execution then proceeds with new followers for `p(1,Z)` and `p(2,Z)`. Since we have failed to detect a fix-point during the last execution of the `fixpoint_check/1` primitive



(4)

	p(1,Z)	
	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE
	p(2,Z)	
	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE

(5)

	p(1,Z)	
▪	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE
	p(2,Z)	
▪	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE

(8)

	p(1,Z)	
	SgFr_state	EVALUATING
▪	SgFr_new_answers	FALSE
	p(2,Z)	
▪	SgFr_state	P_FIXPOINT
▪	SgFr_new_answers	FALSE

(12)

	p(1,Z)	
	SgFr_state	EVALUATING
▪	SgFr_new_answers	TRUE
	p(2,Z)	
	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE
	p(3,Z)	
	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(14)

	p(1,Z)	
	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE
	p(2,Z)	
	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE
	p(3,Z)	
▪	SgFr_state	COMPLETE
▪	SgFr_new_answers	FALSE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(15)

	p(1,Z)	
▪	SgFr_state	NO_FIXPOINT
	SgFr_new_answers	TRUE
	p(2,Z)	
	SgFr_state	P_FIXPOINT
▪	SgFr_new_answers	TRUE
	p(3,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(16)

	p(1,Z)	
▪	SgFr_state	EVALUATING
	SgFr_new_answers	TRUE
	p(2,Z)	
▪	SgFr_state	EVALUATING
▪	SgFr_new_answers	FALSE
	p(3,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(19)

	p(1,Z)	
	SgFr_state	EVALUATING
▪	SgFr_new_answers	FALSE
	p(2,Z)	
▪	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE
	p(3,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

(22)

	p(1,Z)	
▪	SgFr_state	P_FIXPOINT
	SgFr_new_answers	FALSE
	p(2,Z)	
▪	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE
	p(3,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE
	p(4,Z)	
	SgFr_state	COMPLETE
	SgFr_new_answers	FALSE

Figure 6.14: Detecting a partial fix-point not corresponding to completion

for both subgoals, we re-start the computation from the beginning for each follower and reset the `SgFr_new_answers` field to `FALSE`. Next, subgoal `p(2,Z)` is marked as

P_FIXPOINT (step 8 in Fig. 6.14) and we fail. When backtracking we take the second e/2 fact that matches p(1,Z), e(1,3), and, in the continuation, we call p(3,Z) and p(4,Z).

Subgoal p(4,Z) is then marked as COMPLETE (step 11 in Fig. 6.14) and we fail. Next, we obtain a first answer for p(3,Z), p(3,4), and a forth answer for p(1,Z), p(1,4). The fixpoint_check/1 primitive for p(3,Z) then marks p(3,Z) as NO_FIXPOINT (step 12 in Fig. 6.14) and we call again p(4,Z), but no new answers are found. When returning to p(3,Z), we mark it as COMPLETE (step 14 in Fig. 6.14).

Backtracking then sends us back to the follower of p(1,Z), we consume the newly found answer p(1,4), and a forth answer for p(2,Z), p(2,4), is also found. Subgoal p(1,Z) is then marked as NO_FIXPOINT (step 15 in Fig. 6.14) and we fail to the pioneer of p(2,Z). At that point, subgoal p(2,Z) is marked as P_FIXPOINT since we have detected a partial fix-point for it in step 7, but we have also found a new answer for it, p(2,4), in the meantime. This is the typical situation where a partial fix-point does not corresponds to completion. We thus re-start the computation from the beginning for p(2,Z) and reset the SgFr_new_answers field to FALSE (step 16 in Fig. 6.14).

Again, the execution proceeds with new followers for p(1,Z) and p(2,Z). Next, subgoal p(2,Z) is first marked as P_FIXPOINT (step 19 in Fig. 6.14) and then subgoal p(1,Z) is also marked as P_FIXPOINT (step 21 in Fig. 6.14). Backtracking then sends us back to the pioneer of p(2,Z) where now it is marked as COMPLETE (step 22 in Fig. 6.14). Finally, when backtracking to the pioneer of p(1,Z) it is also marked as COMPLETE (step 24 in Fig. 6.14).

Chapter 7

Experimental Results

This chapter presents a detailed performance analysis of the three tabling mechanisms that we have implemented. We start by describing the set of tabled benchmark programs that we have used to assess performance for tabling execution. Next, we measure the performance of our three tabling mechanisms and compare the results with those of YapTab, the built-in tabling engine of the Yap Prolog system. We then discuss several statistics gathered during execution so that the performance results, advantages and weaknesses of each tabling mechanism can be better understood.

7.1 Benchmark Programs

To put the performance results in perspective, we have evaluated our tabling mechanisms against six different versions of the `path/2` program combined with several different configurations of the `edge/2` facts, for a total number of 96 programs. The six different versions of the `path/2` program are presented next in Fig. 7.1. It includes two right recursive, two left recursive and two doubly recursive `path/2` definitions. Each pair has one definition with the recursive clause first and another with the recursive clause last.

Regarding the `edge/2` facts, we have used four main configurations: a binary tree, a pyramid, a loop and a grid configuration. Figure 7.2 shows an example for each configuration of the `edge/2` facts. We have experimented the binary tree configuration with depths 10, 12, 14 and 16; the pyramid and loop configurations with depths 100, 200, 300 and 400; and the grid configuration with 5x5, 10x10, 15x15 and 20x20 nodes.

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

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

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

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

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

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

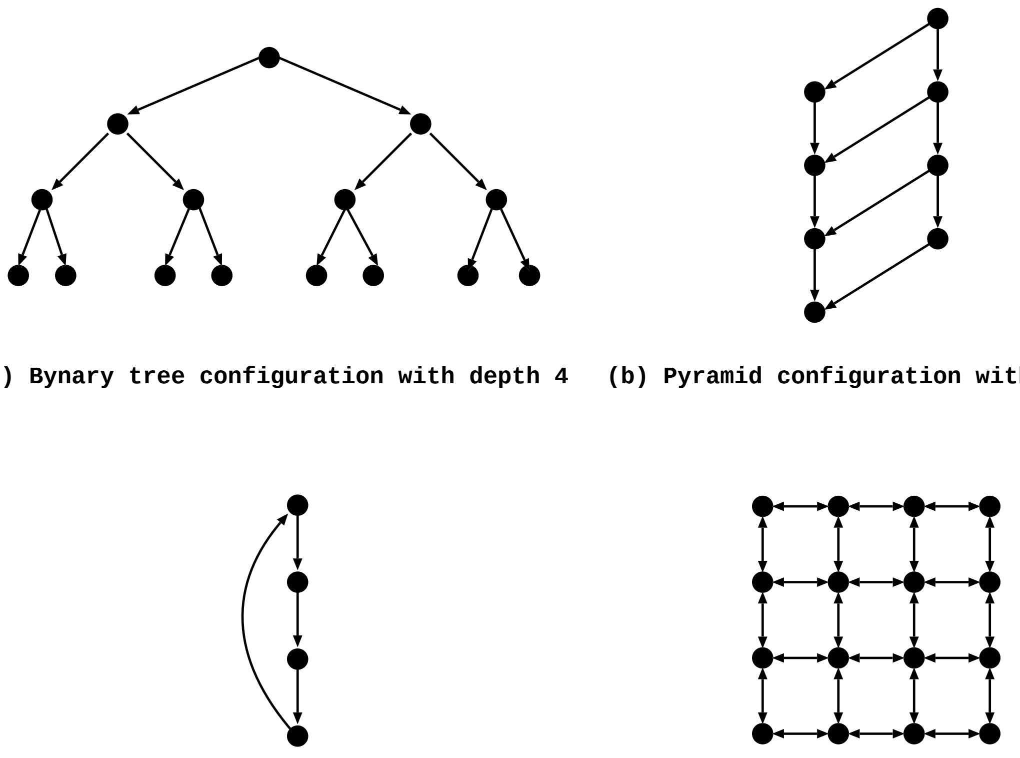
Figure 7.1: The six versions of the `path/2` program

In the loop and grid configurations all pairs of nodes are connected and have an infinite number of paths (if we consider cycles), while in the binary tree and pyramid configurations there are some pairs of nodes that do not have any path. When there is a path between two nodes, for the binary tree the path is unique, while for the pyramid the maximum number of different paths is limited by the depth of the configuration.

All benchmark configurations find all the solutions for the problem. Multiple solutions are computed through automatic failure after a valid solution has been found. Figure 7.3 shows the Prolog code that we used to measure the total running time of each run. The `go/0` predicate is the top query goal and the `run/0` predicate triggers the benchmark execution with the generic query goal `path(X,Z)`.

7.2 Performance Evaluation

The environment for our experiments was a Pentium M 1600MHz processor with 768 MBytes of main memory and running the Linux kernel 2.6.11. All experiments were performed using the YapTab tabling engine based on the Yap Prolog system version 5.1.1 with the default compilation and execution parameters.



(a) Bynary tree configuration with depth 4 **(b) Pyramid configuration with depth 4**

(c) Loop configuration with depth 4 **(d) Grid configuration with 4x4 nodes**

Figure 7.2: The four configurations of the `edge/2` facts

```
go :- statistics(walltime, [Start,_]),
      run,
      statistics(walltime, [End,_]),
      Time is End-Start,
      write('WallTime is '), write(Time), nl.

run :- path(X,Z), fail.
run.
```

Figure 7.3: Prolog code to measure the running time

Tables 7.1, 7.2, 7.3 and 7.4 show the running times, in milliseconds, respectively for the binary tree, pyramid, loop and grid configurations for the six versions of the `path/2` program combined with the four versions of each configuration. Each table measures the performance of our three tabling mechanisms and compares the results with those of YapTab using local scheduling. The overhead over the YapTab execution time is shown in parentheses. Each running time corresponds to the average running time obtained in a set of 3 runs.

Benchmark	**Execution Model**			
	YapTab	**Cont Calls**	**DRA**	**SLDT**
btree_right_first				
depth 10	6	28 (4.67)	19 (3.17)	29 (4.83)
depth 12	35	140 (4.00)	89 (2.54)	146 (4.17)
depth 14	178	664 (3.73)	426 (2.39)	705 (3.96)
depth 16	860	3,112 (3.62)	1,996 (2.32)	3,280 (3.81)
Average		(4.01)	(2.61)	(4.19)
btree_right_last				
depth 10	6	27 (4.50)	19 (3.17)	29 (4.83)
depth 12	37	138 (3.73)	90 (2.43)	149 (4.03)
depth 14	183	657 (3.59)	438 (2.39)	715 (3.91)
depth 16	836	3,096 (3.70)	1,969 (2.36)	3,272 (3.91)
Average		(3.88)	(2.59)	(4.17)
btree_left_first				
depth 10	5	15 (3.00)	18 (3.60)	17 (3.40)
depth 12	26	69 (2.65)	87 (3.35)	85 (3.27)
depth 14	142	340 (2.39)	443 (3.12)	434 (3.06)
depth 16	711	1,667 (2.34)	2,178 (3.06)	2,100 (2.95)
Average		(2.60)	(3.28)	(3.17)
btree_left_last				
depth 10	6	25 (4.17)	18 (3.00)	18 (3.00)
depth 12	25	125 (5.00)	87 (3.48)	86 (3.44)
depth 14	142	612 (4.31)	450 (3.17)	433 (3.05)
depth 16	710	3,021 (4.25)	2,149 (3.03)	2,099 (2.96)
Average		(4.43)	(3.17)	(3.11)
btree_doubly_first				
depth 10	12	114 (9.50)	109 (9.08)	101 (9.25)
depth 12	86	699 (8.13)	664 (7.72)	679 (7.90)
depth 14	509	3,927 (7.72)	3,801 (7.47)	3,816 (7.50)
depth 16	2,746	21,093 (7.68)	20,230 (7.37)	20,281 (7.39)
Average		(8.26)	(7.61)	(8.01)
btree_doubly_last				
depth 10	14	201 (14.36)	110 (7.86)	109 (7.79)
depth 12	83	1,249 (15.05)	665 (8.01)	672 (8.10)
depth 14	503	7,164 (13.96)	3,792 (7.39)	3,820 (7.45)
depth 16	2,751	37,644 (13.68)	20,167 (7.33)	20,327 (7.39)
Average		(14.26)	(7.65)	(7.68)
Total average		(6.28)	(4.53)	(5.06)

Table 7.1: Running times in milliseconds for the binary tree configuration

Benchmark	**Execution Model**			
	YapTab	**Cont Calls**	**DRA**	**SLDT**
pyr_right_first				
depth 100	10	60 (6.00)	35 (3.50)	60 (6.00)
depth 200	41	235 (5.73)	139 (3.39)	237 (5.78)
depth 300	101	530 (5.25)	311 (3.08)	536 (5.31)
depth 400	181	938 (5.18)	552 (3.05)	945 (5.22)
Average		(5.54)	(3.26)	(5.58)
pyr_right_last				
depth 100	10	59 (5.90)	35 (3.50)	61 (6.10)
depth 200	44	236 (5.36)	139 (3.16)	238 (5.41)
depth 300	107	531 (4.96)	314 (2.93)	537 (5.02)
depth 400	182	938 (5.15)	555 (3.05)	945 (5.19)
Average		(5.34)	(3.16)	(5.43)
pyr_left_first				
depth 100	8	27 (3.38)	36 (4.50)	34 (4.25)
depth 200	46	111 (2.41)	140 (3.04)	139 (3.02)
depth 300	116	244 (2.10)	338 (2.91)	319 (2.75)
depth 400	217	439 (2.02)	596 (2.75)	569 (2.62)
Average		(2.48)	(3.30)	(3.16)
pyr_left_last				
depth 100	8	51 (6.38)	36 (4.50)	34 (4.25)
depth 200	45	204 (4.53)	143 (3.18)	137 (3.04)
depth 300	110	461 (4.19)	334 (3.04)	318 (2.89)
depth 400	213	823 (3.86)	601 (2.82)	568 (2.67)
Average		(4.74)	(3.39)	(3.21)
pyr_doubly_first				
depth 100	164	1,803 (10.99)	1,799 (10.97)	1,670 (10.18)
depth 200	1,345	15,145 (11.26)	14,181 (10.54)	13,478 (10.02)
depth 300	4,774	51,552 (10.80)	48,172 (10.09)	45,897 (9.61)
depth 400	11,156	122,324 (10.96)	112,785 (10.11)	107,278 (9.62)
Average		(11.00)	(10.43)	(9.86)
pyr_doubly_last				
depth 100	164	3,522 (21.48)	1,805 (11.01)	1,722 (10.50)
depth 200	1,338	29,184 (21.81)	14,189 (10.60)	13,553 (10.13)
depth 300	4,769	99,989 (20.97)	48,211 (10.11)	46,112 (9.67)
depth 400	11,146	236,481 (21.22)	112,738 (10.11)	107,896 (9.68)
Average		(21.37)	(10.46)	(10.00)
Total average		(8.41)	(5.66)	(6.21)

Table 7.2: Running times in milliseconds for the pyramid configuration

Benchmark	**Execution Model**			
	YapTab	**Cont Calls**	**DRA**	**SLDT**
loop_right_first				
depth 100	6	32 (5.33)	455 (75.83)	150 (25.00)
depth 200	28	122 (4.36)	3,578 (127.79)	1,019 (36.39)
depth 300	70	279 (3.99)	12,293 (175.61)	3,293 (47.04)
depth 400	129	502 (3.89)	28,267 (219.12)	7,796 (60.43)
Average		(4.39)	(149.59)	(42.22)
loop_right_last				
depth 100	5	32 (6.40)	456 (91.20)	153 (36.60)
depth 200	27	123 (4.56)	3,572 (132.30)	1,032 (38.22)
depth 300	70	280 (4.00)	12,247 (174.96)	3,332 (47.60)
depth 400	126	501 (3.98)	28,519 (226.34)	7,839 (62.21)
Average		(4.74)	(156.20)	(46.16)
loop_left_first				
depth 100	5	16 (3.20)	20 (4.00)	18 (3.60)
depth 200	22	67 (3.05)	84 (3.82)	77 (3.50)
depth 300	57	151 (2.65)	187 (3.28)	174 (3.05)
depth 400	118	267 (2.26)	340 (2.88)	322 (2.73)
Average		(2.79)	(3.50)	(3.22)
loop_left_last				
depth 100	5	31 (6.20)	20 (4.00)	19 (3.80)
depth 200	24	123 (5.13)	82 (3.42)	78 (3.25)
depth 300	64	278 (4.34)	187 (2.92)	177 (2.77)
depth 400	116	492 (4.24)	346 (2.98)	324 (2.79)
Average		(4.98)	(3.33)	(3.15)
loop_doubly_first				
depth 100	250	2,636 (10.54)	>1 *day*	33,373 (133.18)
depth 200	2,120	22,160 (10.45)	>1 *day*	526,757 (248.47)
depth 300	6,664	77,122 (11.57)	>1 *day*	2,670,174 (400.69)
depth 400	16,051	180,056 (11.22)	>1 *day*	8,338,538 (519.50)
Average		(10.95)	(*n.a.*)	(325.46)
loop_doubly_last				
depth 100	247	5,181 (20.98)	>1 *day*	33,332 (135.18)
depth 200	2,114	43,048 (20.36)	>1 *day*	527,331 (249.45)
depth 300	6,669	148,270 (22.23)	>1 *day*	2,667,185 (399.94)
depth 400	16,054	348,734 (21.72)	>1 *day*	8,335,909 (519.24)
Average		(21.32)	(*n.a.*)	(325.98)
Total average		(8.19)	(*n.a.*)	(124.36)

Table 7.3: Running times in milliseconds for the loop configuration

Benchmark	**Execution Model**			
	YapTab	**Cont Calls**	**DRA**	**SLDT**
grid_right_first				
5x5 nodes	1	6 (6.00)	21,821 (21,821)	10 (10.00)
10x10 nodes	12	93 (7.75)	$>$ 1 *day*	145 (12.08)
15x15 nodes	76	487 (6.41)	$>$ 1 *day*	753 (9.91)
20x20 nodes	259	1,583 (6.11)	$>$ 1 *day*	2,439 (9.42)
Average		(6.57)	(*n.a.*)	(10.35)
grid_right_last				
5x5 nodes	1	6 (6.00)	20,348 (20,348)	8 (8.00)
10x10 nodes	11	94 (8.55)	$>$ 1 *day*	158 (14.36)
15x15 nodes	78	489 (6.27)	$>$ 1 *day*	792 (10.15)
20x20 nodes	246	1,580 (6.42)	$>$ 1 *day*	2,441 (9.92)
Average		(6.81)	(*n.a.*)	(10.61)
grid_left_first				
5x5 nodes	1	2 (2.00)	3 (3.00)	3 (3.00)
10x10 nodes	9	28 (3.11)	41 (4.56)	40 (4.44)
15x15 nodes	56	138 (2.46)	221 (3.95)	213 (3.80)
20x20 nodes	210	445 (2.12)	749 (3.57)	728 (3.47)
Average		(2.42)	(3.77)	(3.68)
grid_left_last				
5x5 nodes	1	3 (3.00)	3 (3.00)	3 (3.00)
10x10 nodes	9	51 (5.67)	41 (4.56)	40 (4.44)
15x15 nodes	56	265 (4.73)	220 (3.93)	213 (3.80)
20x20 nodes	207	859 (4.15)	748 (3.61)	730 (3.53)
Average		(4.39)	(3.78)	(3.69)
grid_doubly_first				
5x5 nodes	4	44 (11.00)	$>$ 1 *day*	67 (16.75)
10x10 nodes	255	2,637 (10.34)	$>$ 1 *day*	3,948 (15.48)
15x15 nodes	3,300	31,894 (9.66)	$>$ 1 *day*	44,642 (13.53)
20x20 nodes	17,379	180,699 (10.40)	$>$ 1 *day*	242,358 (13.95)
Average		(10.35)	(*n.a.*)	(14.93)
grid_doubly_last				
5x5 nodes	3	80 (26.67)	$>$ 1 *day*	67 (22.33)
10x10 nodes	259	5,113 (19.74)	$>$ 1 *day*	3,990 (15.41)
15x15 nodes	3,377	61,625 (18.25)	$>$ 1 *day*	44,338 (10.15)
20x20 nodes	17,870	349,058 (19.53)	$>$ 1 *day*	241,245 (13.50)
Average		(21.05)	(*n.a.*)	(15.35)
Total average		(8.60)	(*n.a.*)	(9.77)

Table 7.4: Running times in milliseconds for the grid configuration

By observing the results in these tables it is clear that, for these set of experiments, the YapTab tabling engine is the best execution model. On average, it is about 2.5 to 20 times faster than the second best mechanism in each configuration. Regarding our tabling mechanisms, the results indicate that globally the continuation calls execution model obtains better results than the two linear tabling mechanisms. In particular, for programs with more complex dependencies, the loop and grid configurations, it clearly outperforms linear tabling. The results also indicate that globally the continuation calls execution model is comparable to the state-of-the-art YapTab system.

A closer analysis of the results indicates that, on average, the DRA execution model is the best choice when dealing with programs without cycles, as the results for the binary tree and the pyramid configurations show. On the other hand, for the loop and grid configurations the DRA execution model is only competitive if using left recursion. For the right recursive and doubly recursive versions of the `path/2` program it is unsuitable, taking more than one day for most of the configurations.

The results also show that the continuation calls and the SLDT execution models achieve a similar performance for the binary tree and the pyramid configurations. The same can be observed for the loop and grid configurations with left recursion. For the right and doubly recursive versions of the loop and grid configurations, the continuation calls execution model outperforms the SLDT execution model. This behaviour is more clear in the loop configuration. For the right and doubly recursive versions of the loop configuration, the SLDT execution model increases exponentially as the size of the problem also increases. In particular, for the configurations with depth 400, it is about 60 times slower than the YapTab engine for right recursion and about 520 times slower than the YapTab engine for the doubly recursive configurations.

7.3 Performance Analysis

In order to achieve a deeper insight on the behavior of each benchmark program and therefore clarify some of the results presented in the previous section, we next present in Tables 7.5 to 7.28 several statistics gathered during execution for the continuation calls, DRA and SLDT execution models. The rows in these tables have the following meaning:

Answers unique: is the number of non-redundant answers found for tabled subgoals. It corresponds to the total number of answers stored in the table space.

Answers redundant: is the number of redundant answers found for tabled subgoals. A higher number of redundant answers may suggest that the tabling execution model is doing a lot of re-computation to compute fix-points.

Calls unique: is the number of first calls to subgoals corresponding to tabled predicates. It corresponds to the total number of subgoal frames allocated.

Calls repeated: is the number of repeated calls to subgoals corresponding to tabled predicates. A higher number of repeated calls may also suggest that the tabling execution model is doing a lot of re-computation to compute fix-points.

Calls complete: is the number of repeated calls to completed tabled subgoals.

Continuation calls: is the number of continuation calls executed by the primitives `tabled_call/5` and `new_answer/2` in the continuation calls execution model. Remember that the continuation calls are constructed and called using the C language interface, so a higher number of continuation calls corresponds to a proportional cost in the running time.

7.3.1 Right Recursive Configurations

Next we show in Tables 7.5 to 7.12 the statistics gathered for the group of configurations using right recursion.

The statistics clearly show that the behaviour of each execution model is very similar when we use the `path/2` program with the recursive clause first or when we use the version with the recursive clause last. In particular, for the binary tree and pyramid configurations, the statistics are the same (please see Tables 7.5, 7.6, 7.7 and 7.8).

The statistics also show that, for the binary tree and pyramid configurations, the SLDT execution model is the one that performs more re-computation because it finds a higher number of redundant answers and it executes more repeated and completed calls. For these configurations, the continuation calls and the DRA execution models show similar statistics, the small difference observed in the running times results from the cost of constructing and calling the continuation calls using the C language interface for the continuation calls execution model.

For the loop configurations, Tables 7.9 and 7.10, the statistics clearly show that the DRA and SLDT execution models do a lot of re-computation, they find a huge number of redundant answers and they execute a lot of repeated calls. For the grid

btree_right_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	15,366	15,366	15,366
Answers redundant	0	0	15,366
Calls unique	1,023	1,023	1,023
Calls repeated	1,020	508	0
Calls complete	0	512	3,062
Continuation calls	13,324		
Overhead over YapTab	(4.67)	(3.17)	(4.83)
depth 12			
Answers unique	77,830	77,830	77,830
Answers redundant	0	0	77,830
Calls unique	4,095	4,095	4,095
Calls repeated	4,092	2,044	0
Calls complete	0	2,048	12,278
Continuation calls	69,644		
Overhead over YapTab	(4.00)	(2.54)	(4.17)
depth 14			
Answers unique	376,838	376,838	376,838
Answers redundant	0	0	376,838
Calls unique	16,383	16,383	16,383
Calls repeated	16,380	8,188	0
Calls complete	0	8,192	49,142
Continuation calls	344,076		
Overhead over YapTab	(3.73)	(2.39)	(3.96)
depth 16			
Answers unique	1,769,478	1,769,478	1,769,478
Answers redundant	0	0	1,769,478
Calls unique	65,535	65,535	65,535
Calls repeated	65,532	32,764	0
Calls complete	0	32,768	196,598
Continuation calls	1,638,412		
Overhead over YapTab	(3.62)	(2.32)	(3.81)
Average	(4.01)	(2.61)	(4.19)

Table 7.5: Statistics for the btree_right_first configurations

configurations, Tables 7.11 and 7.12, the SLDT model shows comparable numbers to those of the continuation calls model. Regarding the DRA execution model, the statistics obtained for the 5x5 configurations are representative of the re-computation problems of this model in such kind of programs, for example, for the `path/2` version

btree_right_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	15,366	15,366	15,366
Answers redundant	0	0	15,366
Calls unique	1,023	1,023	1,023
Calls repeated	1,020	508	0
Calls complete	0	512	3,062
Continuation calls	13,324		
Overhead over YapTab	(4.50)	(3.17)	(4.83)
depth 12			
Answers unique	77,830	77,830	77,830
Answers redundant	0	0	77,830
Calls unique	4,095	4,095	4,095
Calls repeated	4,092	2,044	0
Calls complete	0	2,048	12,278
Continuation calls	69,644		
Overhead over YapTab	(3.73)	(2.43)	(4.03)
depth 14			
Answers unique	376,838	376,838	376,838
Answers redundant	0	0	376,838
Calls unique	16,383	16,383	16,383
Calls repeated	16,380	8,188	0
Calls complete	0	8,192	49,142
Continuation calls	344,076		
Overhead over YapTab	(3.59)	(2.39)	(3.91)
depth 16			
Answers unique	1,769,478	1,769,478	1,769,478
Answers redundant	0	0	1,769,478
Calls unique	65,535	65,535	65,535
Calls repeated	65,532	32,764	0
Calls complete	0	32,768	196,598
Continuation calls	1,638,412		
Overhead over YapTab	(3.70)	(2.36)	(3.91)
Average	(3.88)	(2.59)	(4.17)

Table 7.6: Statistics for the btree_right_last configurations

with the recursive clause first the DRA model finds 31,797,065 redundant answers and it calls 1,308,990 repeated subgoals!

pyr_right_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	29,900	29,900	29,900
Answers redundant	9,801	9,801	49,502
Calls unique	201	201	201
Calls repeated	396	392	0
Calls complete	0	4	992
Continuation calls	39,105		
Overhead over YapTab	(6.00)	(3.50)	(6.00)
depth 200			
Answers unique	119,800	119,800	119,800
Answers redundant	39,601	39,601	199,002
Calls unique	401	401	401
Calls repeated	796	792	0
Calls complete	0	4	1,992
Continuation calls	158,205		
Overhead over YapTab	(5.73)	(3.39)	(5.78)
depth 300			
Answers unique	269,700	269,700	269,700
Answers redundant	89,401	89,401	448,502
Calls unique	601	601	601
Calls repeated	1,196	1,192	0
Calls complete	0	4	2,992
Continuation calls	357,305		
Overhead over YapTab	(5.25)	(3.08)	(5.31)
depth 400			
Answers unique	479,600	479 600	479,600
Answers redundant	159,201	159,201	798,002
Calls unique	801	801	801
Calls repeated	1,596	1,592	0
Calls complete	0	4	3,992
Continuation calls	636,405		
Overhead over YapTab	(5.18)	(3.05)	(5.22)
Average	(5.54)	(3.26)	(5.58)

Table 7.7: Statistics for the pyr_right_first configurations

7.3.2 Left Recursive Configurations

Tables 7.13 to 7.20 show the statistics gathered for the group of configurations using left recursion.

pyr_right_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	29,900	29,900	29,900
Answers redundant	9,801	9,801	49,502
Calls unique	201	201	201
Calls repeated	396	392	0
Calls complete	0	4	992
Continuation calls	39,105		
Overhead over YapTab	(5.90)	(3.50)	(6.10)
depth 200			
Answers unique	119,800	119,800	119,800
Answers redundant	39,601	39,601	199,002
Calls unique	401	401	401
Calls repeated	796	792	0
Calls complete	0	4	1,992
Continuation calls	158,205		
Overhead over YapTab	(5.36)	(3.16)	(5.41)
depth 300			
Answers unique	269,700	269,700	269,700
Answers redundant	89,401	89,401	448,502
Calls unique	601	601	601
Calls repeated	1,196	1,192	0
Calls complete	0	4	2,992
Continuation calls	357,305		
Overhead over YapTab	(4.96)	(2.93)	(5.02)
depth 400			
Answers unique	479,600	479,600	479,600
Answers redundant	159,201	159,201	798,002
Calls unique	801	801	801
Calls repeated	1,596	1,592	0
Calls complete	0	4	3,992
Continuation calls	636,405		
Overhead over YapTab	(5.15)	(3.05)	(5.19)
Average	(5.34)	(3.16)	(5.43)

Table 7.8: Statistics for the pyr_right_last configurations

By observing the statistics in these tables, it is obvious that the behaviour of the three execution models for left recursive programs is guided by the fact of the programs being left recursive. The three execution models show an identical pattern independently of the configuration being used. The average overhead over the YapTab system is almost

loop_right_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	20,000	20,000	20,000
Answers redundant	200	661,850	196,849
Calls unique	101	101	101
Calls repeated	100	10,001	5,050
Calls complete	0	99	199
Continuation calls	20,000		
Overhead over YapTab	(5.33)	(75.83)	(25.00)
depth 200			
Answers unique	80,000	80,000	80,000
Answers redundant	400	5,313,700	1,453,699
Calls unique	201	201	201
Calls repeated	200	40,001	20,100
Calls complete	0	199	399
Continuation calls	80,000		
Overhead over YapTab	(4.36)	(127.79)	(36.39)
depth 300			
Answers unique	180,000	180,000	180,000
Answers redundant	600	17,955,550	4,770,549
Calls unique	301	301	301
Calls repeated	300	90,001	45,150
Calls complete	0	299	599
Continuation calls	180,000		
Overhead over YapTab	(3.99)	(175.61)	(47.04)
depth 400			
Answers unique	320,000	320,000	320,000
Answers redundant	800	42,587,400	11,147,399
Calls unique	401	401	401
Calls repeated	400	160,001	80,200
Calls complete	0	399	799
Continuation calls	320,000		
Overhead over YapTab	(3.89)	(219.12)	(60.43)
Average	(4.39)	(149.59)	(42.22)

Table 7.9: Statistics for the loop_right_first configurations

the same when comparing the binary tree, pyramid, loop and grid configurations against each particular execution model. In particular, for the DRA and SLDT models, the average overhead over the YapTab system varies between 3.11 and 3.78 for the `path/2` versions with the recursive clause first and last. For the continuation calls

loop_right_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	20,000	20,000	20,000
Answers redundant	200	657,000	201,599
Calls unique	101	101	101
Calls repeated	100	9,902	4,950
Calls complete	0	99	298
Continuation calls	20,000		
Overhead over YapTab	(6.40)	(91.20)	(36.60)
depth 200			
Answers unique	80,000	80,000	80,000
Answers redundant	400	5,294,000	1,473,199
Calls unique	201	201	201
Calls repeated	200	39,802	19,900
Calls complete	0	199	598
Continuation calls	80,000		
Overhead over YapTab	(4.56)	(132.30)	(38.22)
depth 300			
Answers unique	180,000	180,000	180,000
Answers redundant	600	17,911,000	4,814,799
Calls unique	301	301	301
Calls repeated	300	89,702	44,850
Calls complete	0	299	898
Continuation calls	180,000		
Overhead over YapTab	(4.00)	(174.96)	(47.60)
depth 400			
Answers unique	320,000	320,000	320,000
Answers redundant	800	42,508,000	11,226,399
Calls unique	401	401	401
Calls repeated	400	159,602	79,800
Calls complete	0	399	1,198
Continuation calls	320,000		
Overhead over YapTab	(3.98)	(226.34)	(62.21)
Average	(4.74)	(156.20)	(46.16)

Table 7.10: Statistics for the loop_right_last configurations

execution model, it varies between 2.00 and 3.38 for the version with the recursive clause first and between 3.00 and 6.38 for the version with the recursive clause last.

A closer analysis of the results presented in Tables 7.13 to 7.20 shows us that the

grid_right_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	1,250	1,250	1,250
Answers redundant	2,910	31,797,065	12,427
Calls unique	26	26	26
Calls repeated	135	1,308,990	365
Calls complete	0	79	193
Continuation calls	4,000		
Overhead over YapTab	(6.00)	(21,821)	(10.00)
10x10 nodes			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	52,720	*n.a.*	200,578
Calls unique	101	*n.a.*	101
Calls repeated	620	*n.a.*	1,782
Calls complete	0	*n.a.*	851
Continuation calls	72,000		
Overhead over YapTab	(7.75)	(*n.a.*)	(12.08)
15x15 nodes			
Answers unique	101,250	*n.a.*	101,250
Answers redundant	278,430	*n.a.*	1,026,014
Calls unique	226	*n.a.*	226
Calls repeated	1,455	*n.a.*	4,347
Calls complete	0	*n.a.*	1,954
Continuation calls	378,000		
Overhead over YapTab	(6.41)	(*n.a.*)	(9.91)
20x20 nodes			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	899,040	*n.a.*	3,393,067
Calls unique	401	*n.a.*	401
Calls repeated	2,640	*n.a.*	8,214
Calls complete	0	*n.a.*	3,689
Continuation calls	1,216,000		
Overhead over YapTab	(6.11)	(*n.a.*)	(9.42)
Average	(6.57)	(*n.a.*)	(10.35)

Table 7.11: Statistics for the grid_right_first configurations

statistics gathered for the continuation calls, DRA and SLDT execution models are very similar. In particular, for the DRA and SLDT models they are the same when comparing the versions with the recursive clause first and last, the small exception is the number of repeated calls in the DRA model, the `path/2` versions with the recursive

grid_right_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	1,250	1,250	1,250
Answers redundant	2,910	29,149,152	10,394
Calls unique	26	26	26
Calls repeated	135	1,185,756	275
Calls complete	0	79	188
Continuation calls	4,000		
Overhead over YapTab	(6.00)	(20,348)	(8.00)
10x10 nodes			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	52,720	*n.a.*	217,924
Calls unique	101	*n.a.*	101
Calls repeated	620	*n.a.*	1,914
Calls complete	0	*n.a.*	844
Continuation calls	72,000		
Overhead over YapTab	(8.55)	(*n.a.*)	(14.36)
15x15 nodes			
Answers unique	101,250	*n.a.*	101,250
Answers redundant	278,430	*n.a.*	1,089,530
Calls unique	226	*n.a.*	226
Calls repeated	1,455	*n.a.*	4,614
Calls complete	0	*n.a.*	2,084
Continuation calls	378,000		
Overhead over YapTab	(6.27)	(*n.a.*)	(10.15)
20x20 nodes			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	899,040	*n.a.*	3,387,101
Calls unique	401	*n.a.*	401
Calls repeated	2,640	*n.a.*	8,175
Calls complete	0	*n.a.*	3,886
Continuation calls	1,216,000		
Overhead over YapTab	(6.42)	(*n.a.*)	(9.92)
Average	(6.81)	(*n.a.*)	(10.61)

Table 7.12: Statistics for the grid_right_last configurations

clause first execute 3 repeated calls and the versions with the recursive clause last execute only 2.

Regarding the continuation calls execution model, the statistics show that it finds

btree_left_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	8,194	8,194	8,194
Answers redundant	0	7,172	8,194
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	8,194		
Overhead over YapTab	(3.00)	(3.60)	(3.40)
depth 12			
Answers unique	40,962	40,962	40,962
Answers redundant	0	36,868	40,962
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	40,962		
Overhead over YapTab	(2.65)	(3.35)	(3.27)
depth 14			
Answers unique	196,610	196,610	196,610
Answers redundant	0	180,228	196,610
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	196,610		
Overhead over YapTab	(2.39)	(3.12)	(3.06)
depth 16			
Answers unique	917,506	917,506	917,506
Answers redundant	0	851,972	917,506
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	917,506		
Overhead over YapTab	(2.34)	(3.06)	(2.95)
Average	(2.60)	(3.28)	(3.17)

Table 7.13: Statistics for the btree_left_first configurations

a smaller number of redundant answers when compared with the DRA and SLDT models for the `path/2` versions with the recursive clause first, and a higher number of redundant answers for the `path/2` versions with the recursive clause last. The number of continuation calls is also higher for the versions with the recursive clause last, thus

btree_left_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	8,194	8,194	8,194
Answers redundant	6,152	7,172	8,194
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	15,366		
Overhead over YapTab	(4.17)	(3.00)	(3.00)
depth 12			
Answers unique	40,962	40,962	40,962
Answers redundant	32,776	36,868	40,962
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	77,830		
Overhead over YapTab	(5.00)	(3.48)	(3.44)
depth 14			
Answers unique	196,610	196,610	196,610
Answers redundant	163,848	180,228	196,610
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	376,838		
Overhead over YapTab	(4.31)	(3.17)	(3.05)
depth 16			
Answers unique	917,506	917,506	917,506
Answers redundant	786,440	851,972	917,506
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	1,769,478		
Overhead over YapTab	(4.25)	(3.03)	(2.96)
Average	(4.43)	(3.17)	(3.11)

Table 7.14: Statistics for the btree_left_last configurations

justifying the small difference in the running times between the two versions.

pyr_left_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	15,050	15,050	15,050
Answers redundant	4,950	24,651	24,950
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	15,050		
Overhead over YapTab	(3.38)	(4.50)	(4.25)
depth 200			
Answers unique	60,100	60,100	60,100
Answers redundant	19,900	99,301	99,900
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	60,100		
Overhead over YapTab	(2.41)	(3.04)	(3.02)
depth 300			
Answers unique	135,150	135,150	135,150
Answers redundant	44,850	223,951	224,850
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	135,150		
Overhead over YapTab	(2.10)	(2.91)	(2.75)
depth 400			
Answers unique	240,200	240,200	240,200
Answers redundant	79,800	398,601	399,800
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	240,200		
Overhead over YapTab	(2.02)	(2.75)	(2.62)
Average	(2.48)	(3.30)	(3.16)

Table 7.15: Statistics for the pyr_left_first configurations

7.3.3 Doubly Recursive Configurations

Finally, we show in Tables 7.21 to 7.28 the statistics gathered for the group of configurations using doubly recursion.

pyr_left_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	15,050	15,050	15,050
Answers redundant	24,256	24,651	24,950
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	29,801		
Overhead over YapTab	(6.38)	(4.50)	(4.25)
depth 200			
Answers unique	60,100	60,100	60,100
Answers redundant	98,506	99,301	99,900
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	119,601		
Overhead over YapTab	(4.53)	(3.18)	(3.04)
depth 300			
Answers unique	135,150	135,150	135,150
Answers redundant	222,756	223,951	224,850
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	269,401		
Overhead over YapTab	(4.19)	(3.04)	(2.89)
depth 400			
Answers unique	240,200	240,200	240,200
Answers redundant	397,006	398,601	399,800
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	479,201		
Overhead over YapTab	(3.86)	(2.82)	(2.67)
Average	(4.74)	(3.39)	(3.21)

Table 7.16: Statistics for the pyr_left_last configurations

The statistics show that the behaviour of the DRA and SLDT execution models is very similar when we use the `path/2` program with the recursive clause first or last. In particular, for the binary tree and pyramid configurations, the statistics are the same for the SLDT model and very similar for the DRA model (please see Tables 7.21,

loop_left_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	10,000	10,000	10,000
Answers redundant	100	10,100	10,200
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	10,000		
Overhead over YapTab	(3.20)	(4.00)	(3.60)
depth 200			
Answers unique	40,000	40,000	40,000
Answers redundant	200	40,200	40,400
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	40,000		
Overhead over YapTab	(3.05)	(3.82)	(3.50)
depth 300			
Answers unique	90,000	90,000	90,000
Answers redundant	300	90,300	90,600
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	90,000		
Overhead over YapTab	(2.65)	(3.28)	(3.05)
depth 400			
Answers unique	160,000	160,000	160,000
Answers redundant	400	160,400	160,800
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	160,000		
Overhead over YapTab	(2.26)	(2.88)	(2.73)
Average	(2.79)	(3.50)	(3.22)

Table 7.17: Statistics for the loop_left_first configurations

7.22, 7.23 and 7.24). For the loop and grid configurations, the SLDT model shows similar statistics for the first and last versions, while the DRA model do not executes in less than a day even for the smaller configurations (please see Tables 7.25, 7.26, 7.27 and 7.28). For the continuation calls execution model, the statistics clearly show

loop_left_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	10,000	10,000	10,000
Answers redundant	10,000	10,100	10,200
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	19,900		
Overhead over YapTab	(6.20)	(4.00)	(3.80)
depth 200			
Answers unique	40,000	40,000	40,000
Answers redundant	40,000	40,200	40,400
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	79,800		
Overhead over YapTab	(5.13)	(3.42)	(3.25)
depth 300			
Answers unique	90,000	90,000	90,000
Answers redundant	90,000	90,300	90,600
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	179,700		
Overhead over YapTab	(4.34)	(2.92)	(2.77)
depth 400			
Answers unique	160,000	160,000	160,000
Answers redundant	160,000	160,400	160,800
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	319,600		
Overhead over YapTab	(4.24)	(2.98)	(2.79)
Average	(4.98)	(3.33)	(3.15)

Table 7.18: Statistics for the loop_left_last configurations

why the `path/2` versions with the recursive clause first are, on average, about two times faster than the versions with the recursive clause last. The number of redundant answers, repeated calls and continuations calls in the versions with the recursive clause first is about half the number of the correspondent statistics for the last versions.

grid_left_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	625	625	625
Answers redundant	1,455	3,455	3,535
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	625		
Overhead over YapTab	(2.00)	(3.00)	(3.00)
10x10 nodes			
Answers unique	10,000	10,000	10,000
Answers redundant	26,360	62,360	62,720
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	10,000		
Overhead over YapTab	(3.11)	(4.56)	(4.44)
15x15 nodes			
Answers unique	50,625	50,625	50,625
Answers redundant	139,215	328,215	329,055
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	50,625		
Overhead over YapTab	(2.46)	(3.95)	(3.80)
20x20 nodes			
Answers unique	160,000	160,000	160,000
Answers redundant	449,520	1,057,520	1,059,040
Calls unique	1	1	1
Calls repeated	1	3	2
Calls complete	0	0	0
Continuation calls	160,000		
Overhead over YapTab	(2.12)	(3.57)	(3.47)
Average	(2.42)	(3.77)	(3.68)

Table 7.19: Statistics for the grid_left_first configurations

The statistics also show why the continuation calls execution model outperforms the SLDT model in the loop and grid configurations. In particular, for the loop configurations, Tables 7.25 and 7.26, the statistics show that the SLDT model performs a lot of re-computation as the number of redundant answers and repeated calls

grid_left_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	625	625	625
Answers redundant	3,187	3,455	3,535
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	1,170		
Overhead over YapTab	(3.00)	(3.00)	(3.00)
10x10 nodes			
Answers unique	10,000	10,000	10,000
Answers redundant	61,032	62,360	62,720
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	19,640		
Overhead over YapTab	(5.67)	(4.56)	(4.44)
15x15 nodes			
Answers unique	50,625	50,625	50,625
Answers redundant	325,027	328,215	329,055
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	100,410		
Overhead over YapTab	(4.73)	(3.93)	(3.80)
20x20 nodes			
Answers unique	160,000	160,000	160,000
Answers redundant	1,051,672	1,057,520	1,059,040
Calls unique	1	1	1
Calls repeated	1	2	2
Calls complete	0	0	0
Continuation calls	318,480		
Overhead over YapTab	(4.15)	(3.61)	(3.53)
Average	(4.39)	(3.78)	(3.69)

Table 7.20: Statistics for the grid_left_last configurations

report. For example, in the `loop_doubly_first` configuration, the SLDT model finds 13,002,508,000 redundant answers and it calls 32,240,802 repeated subgoals!

btree_doubly_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	15,366	15,366	15,366
Answers redundant	38,888	91,100	93,142
Calls unique	1,023	1,023	1,023
Calls repeated	15,367	1,789	1,534
Calls complete	0	29,198	29,710
Continuation calls	67,578		
Overhead over YapTab	(9.50)	(9.08)	(9.25)
depth 12			
Answers unique	77,830	77,830	77,830
Answers redundant	270,312	610,268	618,454
Calls unique	4,095	4,095	4,095
Calls repeated	77,831	7,165	6,142
Calls complete	0	149,518	151,566
Continuation calls	417,786		
Overhead over YapTab	(8.13)	(7.72)	(7.90)
depth 14			
Answers unique	376,838	376,838	376,838
Answers redundant	1,671,144	3,686,364	3,719,126
Calls unique	16,383	16,383	16,383
Calls repeated	376,839	28,669	24,574
Calls complete	0	729,102	737,294
Continuation calls	2,392,058		
Overhead over YapTab	(7.72)	(7.47)	(7.50)
depth 16			
Answers unique	1,769,478	1,769,478	1,769,478
Answers redundant	9,568,232	20,774,876	20,905,942
Calls unique	65,535	65,535	65,535
Calls repeated	1,769,479	114,685	98,302
Calls complete	0	3,440,654	3,473,422
Continuation calls	12,976,122		
Overhead over YapTab	(7.68)	(7.37)	(7.39)
Average	(8.26)	(7.61)	(8.01)

Table 7.21: Statistics for the btree_doubly_first configurations

7.4 Summary

Globally, the statistics presented in Tables 7.5 to 7.28 confirm that there is a cost in the running times that is proportional to the number of redundant answers, repeated

btree_doubly_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 10			
Answers unique	15,366	15,366	15,366
Answers redundant	77,776	91,100	93,142
Calls unique	1,023	1,023	1,023
Calls repeated	28,691	1,534	1,534
Calls complete	0	29,710	29,710
Continuation calls	119,790		
Overhead over YapTab	(14.36)	(7.86)	(7.79)
depth 12			
Answers unique	77,830	77,830	77,830
Answers redundant	540,624	610,268	618,454
Calls unique	4,095	4,095	4,095
Calls repeated	147,475	6,142	6,142
Calls complete	0	151,566	151,566
Continuation calls	757,742		
Overhead over YapTab	(15.05)	(8.01)	(8.10)
depth 14			
Answers unique	376,838	376,838	376,838
Answers redundant	3,342,288	3,686,364	3,719,126
Calls unique	16,383	16,383	16,383
Calls repeated	720,915	24,574	24,574
Calls complete	0	737,294	737,294
Continuation calls	4,407,278		
Overhead over YapTab	(13.96)	(7.39)	(7.45)
depth 16			
Answers unique	1,769,478	1,769,478	1,769,478
Answers redundant	19,136,464	20,774,876	20,905,942
Calls unique	65,535	65,535	65,535
Calls repeated	3,407,891	98,302	98,302
Calls complete	0	3,473,422	3,473,422
Continuation calls	24,182,766		
Overhead over YapTab	(13.68)	(7.33)	(7.39)
Average	(14.26)	(7.65)	(7.68)

Table 7.22: Statistics for the btree_doubly_last configurations

calls and continuation calls executed during an evaluation.

The statistics also show that the behaviour of the DRA and SLDT execution models is very similar when we use the `path/2` program with the recursive clause first or

pyr_doubly_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	29,900	29,900	29,900
Answers redundant	1,279,146	2,587,596	2,588,192
Calls unique	201	201	201
Calls repeated	29,901	597	400
Calls complete	0	59,597	59,600
Continuation calls	1,338,350		
Overhead over YapTab	(10.99)	(10.97)	(10.18)
depth 200			
Answers unique	119,800	119,800	119,800
Answers redundant	10,448,296	21,015,196	21,016,392
Calls unique	401	401	401
Calls repeated	119,801	1,197	800
Calls complete	0	239,197	239,200
Continuation calls	10,686,700		
Overhead over YapTab	(11.26)	(10.54)	(10.02)
depth 300			
Answers unique	269,700	269,700	269,700
Answers redundant	35,507,446	71,282,796	71,284,592
Calls unique	601	601	601
Calls repeated	269,701	1,797	1,200
Calls complete	0	538,797	538,800
Continuation calls	36,045,050		
Overhead over YapTab	(10.80)	(10.09)	(9.61)
depth 400			
Answers unique	479,600	479,600	479,600
Answers redundant	84,456,596	169,390,396	169,392,792
Calls unique	801	801	801
Calls repeated	79,601	2,397	1,600
Calls complete	0	958,397	958,400
Continuation calls	85,413,400		
Overhead over YapTab	(10.96)	(10.11)	(9.62)
Average	(11.00)	(10.43)	(9.86)

Table 7.23: Statistics for the pyr_doubly_first configurations

last. The continuation calls execution model only shows a similar behaviour for the right recursive programs. For the left or doubly recursive programs, it achieves better results for the versions with the recursive clause first.

pyr_doubly_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	29,900	29,900	29,900
Answers redundant	2,548,491	2,587,596	2,588,192
Calls unique	201	201	201
Calls repeated	59,205	400	400
Calls complete	0	59,600	59,600
Continuation calls	2,636,999		
Overhead over YapTab	(21.48)	(11.01)	(10.50)
depth 200			
Answers unique	119,800	119,800	119,800
Answers redundant	20,856,991	21,015,196	21,016,392
Calls unique	401	401	401
Calls repeated	238,405	800	239,200
Calls complete	0	239,200	800
Continuation calls	21,213,999		
Overhead over YapTab	(21.81)	(10.60)	(10.13)
depth 300			
Answers unique	269,700	269,700	269,700
Answers redundant	70,925,491	71,282,796	71,284,592
Calls unique	601	601	601
Calls repeated	537,605	1,200	1,200
Calls complete	0	538,800	538,800
Continuation calls	71,730,999		
Overhead over YapTab	(20.97)	(10.11)	(9.67)
depth 400			
Answers unique	479,600	479,600	479,600
Answers redundant	168,753,991	169,390,396	169,392,792
Calls unique	801	801	801
Calls repeated	956,805	1,600	1,600
Calls complete	0	958,400	958,400
Continuation calls	170,187,999		
Overhead over YapTab	(21.22)	(10.11)	(9.68)
Average	(21.37)	(10.46)	(10.00)

Table 7.24: Statistics for the pyr_doubly_last configurations

Regarding the running times, the best result in each of the 16 different configurations is always obtained by the continuation calls execution model for the left recursive program with the recursive clause first. The DRA and SLDT execution models also achieve better results for the left recursive programs with the recursive clause first.

loop_doubly_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	1,980,200	*n.a.*	53,157,000
Calls unique	101	*n.a.*	101
Calls repeated	20,001	*n.a.*	515,202
Calls complete	0	*n.a.*	24,850
Continuation calls	2,020,000		
Overhead over YapTab	(10.54)	(*n.a.*)	(133.18)
depth 200			
Answers unique	80,000	*n.a.*	80,000
Answers redundant	15,920,400	*n.a.*	825,294,000
Calls unique	201	*n.a.*	201
Calls repeated	80,001	*n.a.*	4,060,402
Calls complete	0	*n.a.*	99,700
Continuation calls	16,080,000		
Overhead over YapTab	(10.45)	(*n.a.*)	(248.47)
depth 300			
Answers unique	180,000	*n.a.*	180,000
Answers redundant	53,820,600	*n.a.*	4,135,411,000
Calls unique	301	*n.a.*	301
Calls repeated	180,001	*n.a.*	13,635,602
Calls complete	0	*n.a.*	224,550
Continuation calls	54,180,000		
Overhead over YapTab	(11.57)	(*n.a.*)	(400.69)
depth 400			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	127,680,519	*n.a.*	13,002,508,000
Calls unique	401	*n.a.*	401
Calls repeated	320,001	*n.a.*	32,240,802
Calls complete	0	*n.a.*	399,400
Continuation calls	128,320,000		
Overhead over YapTab	(11.22)	(*n.a.*)	(519.50)
Average	(10.95)	(*n.a.*)	(325.46)

Table 7.25: Statistics for the loop_doubly_first configurations

The exception is the DRA model that for the programs without cycles, binary tree and pyramid, is slight better for right recursion. Note that with right recursion, the DRA model calls more tabled subgoals and finds more answers than the left recursive versions, but on the other hand it finds less redundant answers which suggests

loop_doubly_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
depth 100			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	3,960,200	*n.a.*	53,151,949
Calls unique	101	*n.a.*	101
Calls repeated	39,801	*n.a.*	515,101
Calls complete	0	*n.a.*	24,850
Continuation calls	4,019,800		
Overhead over YapTab	(20.98)	(*n.a.*)	(135.18)
depth 200			
Answers unique	80,000	*n.a.*	80,000
Answers redundant	31,840,400	*n.a.*	825,273,899
Calls unique	201	*n.a.*	201
Calls repeated	15,9601	*n.a.*	4,060,201
Calls complete	0	*n.a.*	99,700
Continuation calls	32,079,600		
Overhead over YapTab	(20.36)	(*n.a.*)	(249.45)
depth 300			
Answers unique	180,000	*n.a.*	180,000
Answers redundant	107,640,600	*n.a.*	4,135,365,849
Calls unique	301	*n.a.*	301
Calls repeated	359,401	*n.a.*	13,635,301
Calls complete	0	*n.a.*	224,550
Continuation calls	108,179,400		
Overhead over YapTab	(22.23)	(*n.a.*)	(399.94)
depth 400			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	255,358,774	*n.a.*	13,002,427,799
Calls unique	401	*n.a.*	401
Calls repeated	639,200	*n.a.*	32,240,401
Calls complete	0	*n.a.*	399,400
Continuation calls	256,319,200		
Overhead over YapTab	(21.72)	(*n.a.*)	(519.24)
Average	(21.32)	(*n.a.*)	(325.98)

Table 7.26: Statistics for the loop_doubly_last configurations

that it is doing less re-computation. Independently of the configuration being used, the statistics gathered for the doubly recursive programs always present the higher number of redundant answers, repeated calls and continuation calls. In consequence, the difference between the YapTab system and our three execution models is more

grid_doubly_first	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	1,250	*n.a.*	1,250
Answers redundant	30,160	*n.a.*	97,813
Calls unique	26	*n.a.*	26
Calls repeated	1,251	*n.a.*	2,483
Calls complete	0	*n.a.*	2,020
Continuation calls	32,500		
Overhead over YapTab	(11.00)	(*n.a.*)	(16.75)
10x10 nodes			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	1,980,720	*n.a.*	6,198,496
Calls unique	101	*n.a.*	101
Calls repeated	20,001	*n.a.*	48,411
Calls complete	0	*n.a.*	31,990
Continuation calls	2,020,000		
Overhead over YapTab	(10.34)	(*n.a.*)	(15.48)
15x15 nodes			
Answers unique	101,250	*n.a.*	101,250
Answers redundant	22,681,680	*n.a.*	69,590,648
Calls unique	226	*n.a.*	226
Calls repeated	101,251	*n.a.*	247,436
Calls complete	0	*n.a.*	174,963
Continuation calls	22,882,500		
Overhead over YapTab	(9.66)	(*n.a.*)	(13.53)
20x20 nodes			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	127,683,040	*n.a.*	380,855,132
Calls unique	401	*n.a.*	401
Calls repeated	320,001	*n.a.*	757,946
Calls complete	0	*n.a.*	572,978
Continuation calls	128,320,000		
Overhead over YapTab	(10.40)	(*n.a.*)	(13.95)
Average	(10.35)	(*n.a.*)	(14.93)

Table 7.27: Statistics for the grid_doubly_first configurations

clear when using the doubly recursive versions of the `path/2` program.

The statistics gathered for the DRA execution model also show that for more complex programs, that is, for programs with cycles and with more than an unique call,

grid_doubly_last	**Execution Model**		
	Cont Calls	**DRA**	**SLDT**
5x5 nodes			
Answers unique	1,250	*n.a.*	1,250
Answers redundant	57,410	*n.a.*	98,029
Calls unique	26	*n.a.*	26
Calls repeated	2,341	*n.a.*	2,426
Calls complete	0	*n.a.*	2,025
Continuation calls	60,840		
Overhead over YapTab	(26.67)	(*n.a.*)	(22.33)
10x10 nodes			
Answers unique	20,000	*n.a.*	20,000
Answers redundant	3,908,720	*n.a.*	6,277,826
Calls unique	101	*n.a.*	101
Calls repeated	39,281	*n.a.*	44,165
Calls complete	0	*n.a.*	35,630
Continuation calls	3,967,280		
Overhead over YapTab	(19.74)	(*n.a.*)	(15.41)
15x15 nodes			
Answers unique	101,250	*n.a.*	101,250
Answers redundant	45,084,930	*n.a.*	68,985,035
Calls unique	226	*n.a.*	226
Calls repeated	200,821	*n.a.*	230,440
Calls complete	0	*n.a.*	185,631
Continuation calls	45,385,320		
Overhead over YapTab	(18.25)	(*n.a.*)	(10.15)
20x20 nodes			
Answers unique	320,000	*n.a.*	320,000
Answers redundant	254,467,040	*n.a.*	379,535,750
Calls unique	401	*n.a.*	401
Calls repeated	636,961	*n.a.*	744,203
Calls complete	0	*n.a.*	586,320
Continuation calls	255,420,960		
Overhead over YapTab	(19.53)	(*n.a.*)	(13.50)
Average	(21.05)	(*n.a.*)	(15.35)

Table 7.28: Statistics for the grid_doubly_last configurations

the DRA evaluation mechanism needs to perform a lot of re-computation to detect completion. For these kind of programs the DRA model is unsuitable, taking more than one day for most of the configurations. The running times and statistics gathered for the loop and grid configurations with right or doubly recursion are representative of the re-computation problems of the DRA model in such kind of programs.

The results obtained for the right and doubly recursive versions of the loop and grid configurations also show that the SLDT execution model clearly outperforms the DRA execution model for such problems. However, when compared with the continuation calls execution model, the SLDT model also performs a lot of re-computation. This behaviour is more clear in the loop configuration. Our results thus indicate that the continuation calls execution model is the best choice for more complex problems and that globally it achieves comparable results to those of YapTab, that implements tabling support at the low-level engine.

Chapter 8

Conclusions

This final chapter summarizes the work developed in this thesis. First, we enumerate the main contributions of the thesis, next we suggest some topics for further work, and then we conclude with a final remark.

8.1 Main Contributions

The work described in this thesis can be stated as the design, implementation and evaluation of three different mechanisms to support tabled evaluation in Prolog. A major guideline for our work was to incorporate tabled evaluation into existing Prolog systems by applying source level transformations to a tabled program. The transformed program then uses specific external tabling primitives that provide direct control over the search strategy. To implement the tabling primitives we took advantage of the C language interface of the Yap Prolog system to build external Prolog modules implementing the support for each mechanism. We can distinguish two main modules in each implementation: the module that implements the table space data structures and the module that implements the specific control primitives of each mechanism. We then summarize the main contributions of our work.

The program transformation module. To implement the program transformation step, we have extended the original program transformation module of Ramesh and Chen [RC97] to include the tabling primitives for our mechanisms. According to the tabling mechanism to be used, a tabled logic program is first transformed to include tabling primitives through source level transformations and only then, the resulting program is compiled. No transformation is applied to non-tabled predicates and the performance of Prolog programs without tabling is unaffected. The program transformation module is fully written in Prolog.

The table space external module. The table space uses two levels of tries as proposed by Ramakrishnan *et al.* [RRS$^+$99]: one level stores the subgoal calls, the other the answers. Each different subgoal call to a tabled predicate corresponds to a unique path through the subgoal trie structure. Each unique path through the answer trie nodes corresponds to a different answer to the entry subgoal call. To implement the table space module we took advantage of the tries external module written by Ricardo Rocha to support the efficient execution of Inductive Logic Programming [FRCS03]. The table space module was implemented using the C language interface of the Yap Prolog system.

The tabling primitives external modules. We have implemented tabling primitives for three of the most successful delaying-based and linear tabling mechanisms. We have implemented support for a delayed-based tabling mechanism based on SLG resolution [CW96], that we named tabled evaluation with continuation calls, and for the DRA [GG01] and SLDT [ZSYY00] linear tabling mechanisms. All mechanisms are based on a local scheduling strategy [FSW96] and support tabled evaluation for definite programs, that is, for programs without negation. The tabling primitives were implemented using the C language interface of the Yap Prolog system.

Performance study. We have performed a first and fair comparison study between the three tabling mechanisms implemented. The tabling mechanisms were evaluated against a set of 96 different programs corresponding to right, left and doubly recursive versions of the `path/2` program combined with several different configurations of the `edge/2` facts. During evaluation, we have measured the running times and we have gathered a set of statistics related to the tabled execution. From the results obtained, the following main conclusions can be enumerated.

- In all configurations, the best result was achieved by the continuation calls execution model for the left recursive program with the recursive clause first. The DRA and SLDT execution models also achieved better results for the left recursive programs with the recursive clause first. The exception was the DRA model that, for the configurations without cycles, was slight better for right recursion.

- For the set of programs with complex dependencies, the continuation calls execution model achieved better results than any of the two linear tabling mechanisms. For these kind of programs, linear tabling clearly pays the cost of performing re-computation to compute fix-points. The best linear tabling mechanism was the SLDT execution model. The DRA evaluation mechanism performs a lot of re-computation, taking more than one day for most of the configurations tested.

- Globally, our results also showed that the continuation calls execution model is comparable to the state-of-the-art YapTab system. This is an interesting result because YapTab also implements a delaying-based mechanism based on SLG resolution, uses tries to implement the table space and is implemented on top of the Yap Prolog system. This is thus a first and fair comparison between the approach of supporting tabling at the low-level engine and the approach of supporting tabling by applying source level transformations coupled with tabling primitives. We thus argue that our approach is a good choice to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine.

8.2 Further Work

We then suggest some topics for further work.

Experimentation and portability. The current implementation needs to be tested with a wider range of applications. A more intensive experimentation of each tabling mechanism will certainly found many opportunities for refining and making each implementation more robust and efficient. Further experimentation should also include porting our implementation to other Prolog systems with a C language interface. Currently, we are already working with the group of the Ciao Prolog system [BCC$^+$] to include our implementation of the continuation calls execution model as a module of the Ciao system.

A new linear tabling mechanism. In computations with multiple calls containing recursive calls, the DRA and SLDT execution models execute the same computations several times until reaching a fix-point. One advantage of the DRA execution model is that only the looping alternatives are recomputed. In the SLDT execution model, repeated calls avoid executing all the alternatives and execute from the backtracking point of the former repeated call. Starting from these observations and from the performance results obtained in this thesis, we are already working on a new proposal that tries to combine the best features of both mechanisms in order to produce a more robust and competitive linear tabling mechanism.

Support for negation. A wide range of applications that use tabling require the expressiveness granted by the possibility of manipulating negative subgoals. Extending our mechanisms to efficiently support negation will certainly be one major step forward to make them usable by a larger community.

8.3 Final Remark

Through this research we have described the basic execution models for three of the most successful delaying-based and linear tabling mechanisms, we have showed how a tabled program is transformed to include specific tabling primitives for each tabling mechanism, and we have presented all the details for implementing each mechanism as an external Prolog module in the Yap Prolog system. The results obtained show us that our approach is a good choice to incorporate tabling into any Prolog system. We hope that the work developed in this thesis will serve as an inspiration to others and be a resource for further improvements and research in this area.

References

[BCC$^+$] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. *Ciao Prolog System Manual.* Available from `http://clip.dia.fi.upm.es/Software/Ciao`.

[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.

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3 & 4):255–299, 1991.

[Car90] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine.* PhD thesis, The Royal Institute of Technology, 1990.

[CM94] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer-Verlag, fourth edition, 1994.

[CW96] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

[Die87] S. Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, Department of Computer Science, State University of New York, 1987.

[DS98] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1490 in LNCS, pages 21–35. Springer-Verlag, 1998.

[DS00] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, 2000.

[FD92] C. Fan and S. Dietrich. Extension Table Built-Ins for Prolog. *Software Practice and Experience*, 22(7):573–597, 1992.

[FRCS03] N. Fonseca, R. Rocha, R. Camacho, and F. Silva. Efficient Data Structures for Inductive Logic Programming. In *International Conference on Inductive Logic Programming*, number 2835 in LNAI, pages 130–145. Springer-Verlag, 2003.

[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.

[Gra96] P. Graf. Term Indexing. Number 1053 in LNAI. Springer-Verlag, 1996.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[McC92] W. W. McCune. Experiments with Discrimination - Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[Mic68] D. Michie. Memo Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[Ohl90] H. J. Ohlbach. Abstraction Tree Indexing for Terms. In *European Conference on Artificial Intelligence*, pages 479–484. Pitman Publishing, 1990.

[RC97] R. Ramesh and W. Chen. Implementation of Tabled Evaluation with Delaying in Prolog. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):559–574, 1997.

[RFS05] R. Rocha, N. 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.

[Rob65] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Roy90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

[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.

[RSS$^+$97] P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. In *International Conference on Logic Programming and Non-Monotonic Reasoning*, number 1265 in LNCS, pages 431–441. Springer-Verlag, 1997.

[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.

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

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog*. 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.

[SS06] Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer-Verlag, 2006.

[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.

[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.

[Vie89] L. Vieille. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

[War77] D. H. D. Warren. *Applied Logic – Its Use and Implementation as a Programming Tool.* PhD thesis, Edinburgh University, 1977.

[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.