Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. A. Bonner, for his guidance, advice and supervision towards the completion of this thesis. I am also indebted to my second reader, Prof. A. O. Mendelzon, for his useful comments on the weaknesses of the thesis.

Dimitris Plexousakis and Thodoros Topaloglou spent some of their valuable time proofreading this thesis and suggesting technical improvements.

My lab and office-mates, George, Dimitra, Makis, Anna, Roni, Kari, Robert, Paul, Weijun, and Franscois provided a peaceful and stimulating working environment.

This thesis would have been impossible to complete, without the generous financial support of the Computer Science Department.

I am most indebted to my barbecue and bouzouki night colleagues, and friends for life, George Charalabopoulos – together with whom I strove towards the accomplishment of our goals – Nick Liodakis, Dimitris Plexousakis, Voula Dakanali Plexousaki, and Thodoros Topaloglou, for being a constant source of support and advise.

Last, but not least, I thank my parents, for I owe them my intellectual and moral stature.

Contents

Chapter 1

Introduction

1.1 Motivation

Modifications of first-order theories [End72] are referred to as *updates*. Although query languages have been studied extensively, updates have only recently become the object of thorough investigation. Commercial systems provide limited support for updates. For example, relational systems can express queries and simple insertions and deletions, but they resort to higher-level programming languages with embedded SQL to execute complex updating procedures (hereafter referred to as *transactions*). Prolog is an early example of a commercial logic programming language which supports updates. However, updates in Prolog are not logical operators, and their semantics and behaviour are not well defined. Research efforts have concentrated on specifying update semantics (e.g., [RFV83, Win88, KM90, Rei92]) and *programming* updates, i.e., specifying composite updates in terms of simpler ones (e.g., [AV88, NK88, NT89, BK93]). Several of these efforts propose declarative programming languages augmented with procedural features as transaction specification tools.

Transaction Logic (abbreviated $\mathcal{T}_\mathcal{R}$) extends first-order logic with advanced transaction specification and execution capabilities [BK93]. Unlike other logics, $\mathcal{T}_\mathcal{R}$ is a general logic designed to suit a large domain of applications, especially Relational [Ull89a] and

Deductive Databases [GMN84], Logic Programming [Llo87], and AI [GN87]. One can *specify* and *execute* transactions in $\mathcal{T}_\mathcal{R}$ in a *declarative* fashion. Increased expressiveness and functionality results from the declarative integration of *procedural* operators. The functionality of $\mathcal{T}_\mathcal{R}$ includes transaction definition, database queries and updates, bulk updates, committed updates, conditions on database state transformations, sequential execution, deterministic and nondeterministic transactions, and several other features, which are beyond the scope of this thesis. Its characteristic feature is that the evaluation of transactions is done on *paths*, i.e., sequences of database states. This approach accounts for a uniform treatment of queries and updates.

Like classical logic, $\mathcal{T}_\mathcal{R}$ has a "Horn" version, Horn–$\mathcal{T}_\mathcal{R}$, which allows users to program transactions. The focus of our work is transaction execution in Horn–$\mathcal{T}_\mathcal{R}$. We take into account updates, non-deterministic transactions and sequential execution.

1.2 Objectives

A Horn–$\mathcal{T}_\mathcal{R}$ program can be interpreted in a number of ways. Bonner and Kifer [BK93] define transaction execution with respect to *all* possible interpretations of a given program. An equivalent definition can be given with respect to a certain interpretation, which is taken to be the *intended meaning* of a given program.

In this work, the first objective is the formal specification of the intended meaning of Horn–$\mathcal{T}_\mathcal{R}$ programs. Guided by the work on the specification of semantics of classical Horn programs [Llo87], we approach the problem from two points of view. From a model-theoretic point of view, we define the intended meaning of a Horn–$\mathcal{T}_\mathcal{R}$ program, P, to be a specific model of P, called the "least Herbrand model" of P. From a proof-theoretic point of view, we give a fixpoint characterization of Horn–$\mathcal{T}_\mathcal{R}$ programs. Finally, we show the equivalence of the aforementioned semantics. Our results are natural extensions of existing results for classical Horn logic.

The second objective is to investigate methods for executing transactions in Horn–$\mathcal{T}_\mathcal{R}$, that comply with the model-theoretic and proof-theoretic semantics. Since Horn–$\mathcal{T}_\mathcal{R}$

is an extension of classical Horn logic, a natural approach is to extend existing query processing methods. Our intention is to explore the feasibility of this approach.

Query processing has attracted a considerable amount of research in the last few years both in Logic Programming [Llo87] and Deductive Databases [GMN84]. The focus has been recursive queries, since non-recursive queries can be handled by standard relational algebra methods. A coarse characterization of the methods in the literature divides them into top-down, bottom-up, and combinations of these. Unfortunately, little work has been done on the execution of updating transactions (e.g., [Bra91, NT89, BLV93]). Bottom-up evaluation of updating transactions is totally unexplored, and simplified approaches are taken for top-down evaluation. For example, Prolog (e.g., [Bra91]) executes simple updates, using the "assert" and "retract" operators. The semantics of these operators, however, is not logical, and their interaction with other logical operators is not clear. Furthermore, there is no logical account of the order of update operations, and an update cannot be rolled back once executed. LDL [NT89] uses only bulk updates, since it is designed for database applications. Its major drawback is that the model theory assumes commutativity of updates. On this assumption, execution of a sequence of goals proceeds in two phases, executing first the queries and then the updates. EKS [BLV93] takes a pragmatic approach, using Prolog as a host language. As a result, EKS inherits Prolog's problems. Moreover, all existing commercial or experimental systems execute transactions on database states. Contrary to this, $\mathcal{T}_\mathcal{R}$ executes transactions on sequences of database states.

We show that evaluation methods for classical Horn logic can be extended naturally to Horn–$\mathcal{T}_\mathcal{R}$, thanks to the path semantics and the clean model theory of the logic. We deal with both bottom-up and top-down evaluation methods. The main issues we consider are *soundness*, *completeness*, *termination* and *efficiency*.

First, we devise a bottom-up evaluation method that adheres to path execution, based on the fixpoint semantics of the logic. We then show that bottom-up execution must be augmented with top-down functionality to handle updates realistically and effectively. The final solution we propose is a combination of rule rewriting techniques [Ull89b], for adding top-down control, with bottom-up evaluation. Bottom-up execution is appropriate for deductive databases and deterministic execution.

Unfortunately, bottom-up evaluation performs poorly for non-deterministic transactions, because it computes the totality of the least fixpoint. Non-deterministic execution is handled better by top-down evaluation. For top-down processing, we extend SLD resolution [AvE82] to $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ resolution, and we prove its soundness and completeness. Then, we add bottom-up functionality to $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ resolution, resulting in $\mathrm{SLD\text{-}AL}^{\mathcal{T}_\mathcal{R}}$ resolution. $\mathrm{SLD\text{-}AL}^{\mathcal{T}_\mathcal{R}}$ resolution extends SLD-AL resolution [Vie87, Vie88, Vie89]. Finally, we show that $\mathrm{SLD\text{-}AL}^{\mathcal{T}_\mathcal{R}}$ resolution is a framework for transaction execution with great potential.

1.3 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2 is a short overview of Horn logic and evaluation methods for Horn logic.

Chapter 3 describes Transaction Logic. It defines general $\mathcal{T}_\mathcal{R}$ and Horn–$\mathcal{T}_\mathcal{R}$ programs, presents the model theory of the logic, defines transaction execution, and discusses executional requirements.

Chapter 4 defines the semantics of Horn–$\mathcal{T}_\mathcal{R}$ programs from a model-theoretic and a proof-theoretic point of view, and shows their equivalence.

Chapters 5 and 6 are devoted to transaction execution. Chapter 5 investigates bottom-up execution. It shows that pure bottom-up execution is unrealistic, and shows how it can be augmented with top-down functionality to yield a realistic approach. The proposed method leaves a few of the problems open, but it is the first serious work on processing rules with updates in the body bottom-up. Chapter 6 explores top-down execution. It defines two theoretical frameworks for transaction execution, namely $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ and $\mathrm{SLD\text{-}AL}^{\mathcal{T}_\mathcal{R}}$ resolution, and gives soundness and completeness results. Finally, it proposes an evaluation strategy which is complete and terminating for a large class of transactions.

Chapter 7 draws conclusions and provides guidelines for future research.

Chapter 2

Overview of Classical Horn Logic

Classical Horn logic [Llo87] emerged from Kowalski [Kow74] and Colmerauer's [CKRP73] idea that logic can be used as a programming language, and since then has served as the basis for many declarative programming languages. This has been achieved through the assignment of procedural semantics to Horn logic, which allows users to execute programs. Section 2.1 gives the basic syntax and informal semantics of Horn logic. Section 2.2 describes three different approaches for query evaluation, namely bottom-up, top-down and combinations of these. It also makes a short comparison of these approaches. An elaborate presentation of evaluation methods appears in [Ull89b], and a detailed comparison of certain methods in [BR88].

2.1 Notation and Informal Semantics

In what follows we assume that the reader is familiar with the basics of first-order logic [End72]. [Llo87] is a detailed exposition of Horn logic.

Horn Clauses

A *clause* (or *rule*) is a pair of sets of atomic formulae written as

$$\alpha_1 \ \& \ \ldots \ \& \ \alpha_m \leftarrow \beta_1 \ \& \ \ldots \ \& \ \beta_n, \quad m, n \geq 0.$$

α_1 & ... & α_m is the *conclusion* (or *head*) of the clause. β_1 & ... & β_n is the *premise* (or *body*) of the clause. If p is an n-place predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula*, or *atom*. Variables and constants are *terms*. If f is an n-place function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A clause,

$$\alpha_1 \,\&\, \ldots \,\&\, \alpha_m \leftarrow \beta_1 \,\&\, \ldots \,\&\, \beta_n,$$

is equivalent to

$$\forall X_1, \ldots, \forall X_k(\alpha_1 \vee \ldots \vee \alpha_m \vee \neg\beta_1 \vee \ldots \vee \neg\beta_n),$$

where $X_1, \ldots, X_k$ are all the variables occurring in these atoms. This is interpreted as

$$\text{for all } X_1, \ldots, X_k, \;\; \alpha_1 \text{ or } \ldots \text{ or } \alpha_m \quad \text{if} \quad \beta_1 \text{ and } \ldots \text{ and } \beta_n,$$

A *definite clause* is one where $m = 1$. A *negative clause* (or *definite goal*) is one where $m = 0$ and $n > 0$. A *unit clause* is one where $m = 1$ and $n = 0$. An *empty clause* is one where $n = m = 0$. Empty clauses are denoted as $\Box$. *Horn clauses* are definite or negative clauses. A *definite program* is a set of definite clauses. A negative clause is considered as a goal to be executed, and an empty clause as a contradiction. A definite clause

$$\alpha \leftarrow \beta_1 \,\&\, \ldots \&\, \beta_n$$

is considered as a *procedure definition*, where α is the procedure name, and β_1 & ...& β_n is its body. A definite program is understood as the conjunction of its clauses. A program is run by giving it an initial *goal*. If $\leftarrow \beta_1$ & ... & β_n is a goal, then each β_i is regarded as a *procedure call*.

Substitutions

A *substitution*, θ, is a finite set of the form $\{X_1/t_1, \ldots, X_n/t_n\}$, where each X_i is a variable, each t_i is a term distinct from X_i, and the variables $X_1, \ldots, X_n$ are distinct. Each element X_i/t_i is called a *binding* for X_i. θ is called a *ground substitution*, if each t_i is a ground term. The *identity substitution* is given by the empty set. Let $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ and $\sigma = \{Y_1/s_1, \ldots, Y_m/s_m\}$ be substitutions. Then, the *composition*, $\theta\sigma$, of θ and σ is the substitution obtained from the set

$$\{X_1/t_1\sigma, \ldots, X_n/t_n\sigma, Y_1/s_1, \ldots, Y_m/s_m\}$$

by deleting any binding $X_i/t_i\sigma$ for which $X_j = t_j\sigma$, and deleting any binding Y_i/s_i for which $Y_j \in \{X_1, \ldots, X_n\}$.

If θ is a substitution and α is an expression, the result is written as $\alpha\theta$, and is called an *instance* of α; α is said to be *more general* than $\alpha\theta$. A *variant* of an expression, α, is an expression that differs from α at most in the names of its variables. Let $\alpha \equiv \beta$ denote that α and β are the same sequence of symbols. If $\alpha_1, \ldots, \alpha_n$ are expressions, a substitution, θ, is called a *unifier* of $\alpha_1, \ldots, \alpha_n$, if $\alpha_1\theta \equiv \ldots \equiv \alpha_n\theta$. A substitution, θ, is called the *most general unifier* (abbreviated *mgu*) of $\alpha_1, \ldots, \alpha_n$, if for every other unifier, σ, of $\alpha_1, \ldots, \alpha_n$, there is a substitution, η, such that $\sigma = \theta\eta$.

Interpretations and Models

An *interpretation* (or *structure*) of a first-order language, $\mathcal{L}$, is a triple $\langle D_{\mathcal{L}}, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$, where:

- $D_{\mathcal{L}}$ is a non-empty set of symbols, called the *domain* of the interpretation.
- $I_{\mathcal{F}}$ is a mapping that assigns to each constant in $\mathcal{L}$ an element in $D_{\mathcal{L}}$, and to each n-place function symbol in $\mathcal{L}$ a mapping $D_{\mathcal{L}}^n \rightarrow D_{\mathcal{L}}$.
- $I_{\mathcal{P}}$ is a mapping that assigns to each n-place predicate symbol in $\mathcal{L}$ a mapping $D_{\mathcal{L}}^n \rightarrow \{true, false\}$.

The truth of a formula, α, in a language, $\mathcal{L}$, is specified with respect to an interpretation of $\mathcal{L}$. An interpretation, I, is a *model* of α if α is true with respect to I. I is a model of a set, S, of closed formulae, if I is a model of each formula in S.

Horn logic considers only *Herbrand structures*, i.e., structures with the following properties.

1. The domain of a Herbrand structure for a first-order language, $\mathcal{L}$, is the *Herbrand universe*, $U_{\mathcal{L}}$, of $\mathcal{L}$. The Herbrand universe of $\mathcal{L}$ is the set of all ground terms, which can be formed out of the constants and function symbols appearing in $\mathcal{L}$.

2. If f is an n-place function symbol in $\mathcal{L}$, then f is assigned the mapping $t_1, \ldots, t_n \rightarrow f(t_1, \ldots, t_n)$ from $(U_{\mathcal{L}})^n$ to $U_{\mathcal{L}}$. In particular, constants in $\mathcal{L}$ are assigned themselves in $U_{\mathcal{L}}$.

Each Herbrand model of a definite program, P, is a possible interpretation of the meaning of P. The intersection of a set of Herbrand models for a definite program, P, is a model of P. This property is the *model intersection property* for definite programs. The intersection of *all* Herbrand models of P is called the *least Herbrand model* of P, and it is taken to be the *intended meaning* of P.

2.2 Evaluation Methods for Classical Horn Logic

A *logic database* or *logic program* is an enumerable set of definite clauses. A *deductive database* is a finite and function-free logic program. It is a convention to partition the set of clauses into a set of *facts*, which are stored in *base predicates* (the Extensional Database or EDB), and a set of *derived predicates* (the Intensional Database or IDB). The derived predicates are those that appear in clause heads. An instance of an EDB predicate is considered true if and only if it is stored as a tuple in the corresponding relation. Given a logic program, P, and a query (i.e., a list of goals), represented as a negative clause, $\leftarrow \alpha$, query evaluation involves a proof of the unsatisfiability of $P \cup \{\leftarrow \alpha\}$. The proof procedure returns a set of substitutions for the variables in α (the *answers* to the query). A substitution θ is a *correct answer* if and only if $P \cup \{\leftarrow \alpha\theta\}$ is unsatisfiable or equivalently, if and only if $\alpha\theta$ is a logical consequence of P. If α is ground, then the set of answers is simply a singleton set containing an empty substitution (interpreted as "yes") or the empty set (interpreted as "no"). Thus, a query processor for logic programs receives as input a program P and a query, and it generates correct answers for the query with regard to P.

In the sequel, we outline the existing computational approaches for query processing.

2.2.1 Bottom-Up Evaluation

The bottom-up approach (also known as *forward chaining*) corresponds to the fixpoint semantics of Horn logic [vEK76, AvE82, Llo87]. The computed fixpoint corresponds to that portion of the program that is relevant to the query. Computation starts by selecting the relevant clauses and predicates.[1]

Bottom-up methods are iterative. Each iteration step, n, proceeds as follows: Each predicate denotes an extensional or intensional database relation. A subgoal, $p(a_1, \ldots, a_k)$, is true, if and only if the tuple $(a_1, \ldots, a_k)$ is in the relation for p. Each subgoal in a rule body is associated with a possibly empty relation, which consists of those tuples that make the subgoal true. Each rule body is also associated with a relation, namely the *join* of all the relations associated with its subgoals. The relation associated with a rule head is the *projection* of the body relation on the variables that appear in the head. The relation associated with an IDB predicate, p, is the *union* of the relations for p over all rules with p in the head.

The facts derived in the n-th iteration are fed into the rule bodies in the $(n+1)$-st iteration to generate new tuples for the head relations. The iterative process terminates when no new facts are generated for any relation. A final stage selects the facts that comprise the answer to the query.

The method just described may generate the same facts several times using the same derivation paths. It is thus called Naive Evaluation [Ull89a]. A more efficient method uses the differentials of the sets of facts to generate new facts, and is called Semi-naive Evaluation [Ull89a].

[1] This selection cannot be made optimally. Specifically, we cannot decide if there is a subset of a set of given rules that computes the same answer for a given query as the original set [Shm87]. In fact, relevance is a runtime notion, and for each query, relevant data are those produced during a *top-down* evaluation of the goal. As a result, we select potentially relevant rules.

2.2.2 Top-Down Evaluation

The top-down approach (also known as *backward chaining*) is based on the idea that a query must be proved to be a logical consequence of a given program. Thus, only the portion of the minimal model that contributes to the proof is constructed. According to this approach, a rule, $\alpha \leftarrow \beta$, is regarded as a *procedure definition*, where α is the name of the procedure, and β is its body. A query is considered to be a list of procedure calls. The notion of execution is materialized by Robinson's *resolution principle* [Rob65]. A goal is resolved with a clause whose head *unifies* (i.e., matches) with the goal, that is, it is replaced with the body of the clause by applying the unifying substitution to the rule and to the goal list. The process is repeated until the goal list becomes empty (success), or no new unifying rules can be found (failure). The most well-known framework for top-down evaluation methods is SLD resolution [AvE82, Llo87].

Intuitively, a top-down answer procedure involves the construction of a tree-like *search space* and a method to traverse it seeking answers for the query. The procedure manipulates tuples or small sets of tuples. The most common search spaces are SLD-trees [AvE82, Llo87] and Rule/Goal trees [Ull89b].

2.2.3 Top-Down vs Bottom-Up

In the following we make a brief comparison of top-down and bottom-up methods. A detailed discussion appears in [Ull89b].

Completeness and Termination

Bottom-up evaluation always converges to the least fixpoint, and it terminates whenever the latter is finite. Contrary to this, top-down methods may get stuck in loops and not terminate, or may fail to return a complete set of answers. However, a top-down method may be shown superior to bottom-up evaluation if the least model is infinite, and the query is safe (i.e., the set of answers is finite). Infinite least models arise because of function symbols or infinite relations (which may represent computed functions). In these cases, bottom-up evaluation is complete, but does not terminate. Goal-oriented, top-down computation succeeds to find all answers and terminate in many cases. Nevertheless, in deductive databases bottom-up evaluation beats top-down methods, with respect to completeness and termination.

Efficiency

Top-down methods are goal-oriented, and can therefore *focus on relevant data*, i.e., facts that actually contribute to the answer of the query, thus avoiding unnecessary computation. Bottom-up evaluation, however, tends to generate a significant number of irrelevant facts. Top-down methods can make use of efficient, stack-based memory management, where the space used is proportional to the longest stack of subgoals. Bottom-up evaluation uses space proportional to the number of derived facts, which can be substantially larger than the longest stack. However, redundant computation is present in top-down methods, in the form of recomputing previous subgoals. Run-time optimization techniques are adopted in some cases to cope with this kind of redundancy.

A nice property of bottom-up evaluation, which makes it suitable for deductive databases, is the ability to perform massive joins. This is not possible in most top-down methods, since they tend to manipulate tuples, or small sets of tuples, not entire relations. The difference in performance becomes considerably noticeable when the relations involved in joins are large, or if the top-down method computes the same joins repeatedly.

2.2.4 Combined Evaluation

Several methods in the literature combine top-down and bottom-up evaluation to benefit from the advantages of both approaches. They can be divided into compile-time and run-time methods.

Compile-time methods

Compile-time methods transform a logic program with respect to a query, so that a more efficient program is generated and evaluated in a bottom-up fashion. Transformation involves rule rewriting, based on a *sideways information passing strategy* (sip) and a control strategy [BR87, vG86]. Sip determines how information (variable bindings) about some subgoals in a rule is used to evaluate other subgoals in the rule, and the control strategy implements this (possibly using other optimizations as well). The objective is to simulate the top-down evaluation of the original program through bottom-up evaluation, so that computation is limited to relevant data.

The fundamental method for general Horn programs is the Magic Sets method [RLK86, BMSU87, Ull89b], and its extensions [Ull89b, Sek89, Ram88]. A plethora of specialized methods for increased performance appears in the literature [BMSU87, Ull89b].

Run-time methods

Run-time methods are also referred to as *memoing* methods [MS81, Loz85, TS86, Die87, Vie87, Vie88, Vie89]. The concept is to remember (*memo*) previously computed answers and reuse them when needed. The search space consists of a network of producers and consumers. The computation proceeds in a top-down fashion, resolving goals and producing answers. A instance of a previously processed goal consumes the facts generated by that goal and uses them in inferences.

Chapter 3

Transaction Logic Programming

This chapter introduces Transaction Logic. We outline the semantics of the logic and give the main definitions. The interested reader is referred to [BK93] for an in-depth development. Section 3.1 is brief introduction to the main characteristics of $\mathcal{T}_{\mathcal{R}}$. Section 3.2 specifies the structure and the syntax of $\mathcal{T}_{\mathcal{R}}$ programs. Section 3.3 introduces *Horn–$\mathcal{T}_{\mathcal{R}}$*, an important fragment of the logic, which is an extension of classical Horn-clause logic. The class of Horn–$\mathcal{T}_{\mathcal{R}}$ programs is the focus of our work. Section 3.4 presents the model theory of the logic. In particular, it introduces *path structures*, the counterparts of first-order semantic structures, and gives the definition of satisfaction in $\mathcal{T}_{\mathcal{R}}$. Section 3.5 defines transaction execution. Finally, Section 3.6 discusses our requirements for transaction execution.

3.1 Introduction

$\mathcal{T}_{\mathcal{R}}$ is an extension of first-order logic (FOL) designed to account for the phenomenon of state changes in logic programs and databases. Like FOL, $\mathcal{T}_{\mathcal{R}}$ has a "Horn" version, with both declarative and procedural semantics, which allows transaction *programming*. Its functionality includes database queries and updates, serial execution, and non-deterministic execution.

Syntactically, $\mathcal{T}_\mathcal{R}$ extends FOL with a binary operator, $\otimes$, called *serial conjunction*. Informally, the formula $\alpha \otimes \beta$ means: "First execute transaction α, and then β." This operator can be combined arbitrarily with other classical operators of logic to build composite formulae.

Semantically, evaluation of formulae in $\mathcal{T}_\mathcal{R}$ differs from classical evaluation in that it is made on *paths*, i.e., sequences of database states. The reason is that a database changes states when updates occur during transaction execution. A transaction transforms a database from an initial state to a final state, passing through a sequence of intermediate states. The transaction $\alpha \otimes \beta$ cannot be executed successfully unless the final database state of the execution of transaction α is the initial state for the execution of β.

Although $\mathcal{T}_\mathcal{R}$ is concerned with the specification of transactions, it is *not* concerned with the specification of the semantics of "elementary updates". Such updates are defined through logical axioms provided by users. In this way, different update semantics can be incorporated into $\mathcal{T}_\mathcal{R}$. Composite updates are defined in terms of elementary updates. This approach accounts for the clean model and proof theory of the logic. A formal overview of the logic follows.

3.2 $\mathcal{T}_\mathcal{R}$ programs

3.2.1 Program Components

A logic program in $\mathcal{T}_\mathcal{R}$ is a triplet $(\mathcal{B}, \mathbf{P}, \mathbf{D})$, where:

- $\mathbf{P}$ is a *transaction base*, i.e, a set of *transaction formulae*. Transaction formulae are coded by users and define transactions.

- $\mathbf{D}$ is a database, i.e., a set of classical first-order formulae. Special cases of $\mathbf{D}$ may be relational or deductive databases and logic programs.

- $\mathcal{B}$ is a *transition base*, which is a set of *elementary state transitions*. Each elementary state transition is a logical formula defining a state change.

Transactions in **P** may update **D**. This is the only updatable component of a $\mathcal{T}_\mathcal{R}$ program. State changes defined through elementary state transitions are called *elementary updates*. $\mathcal{B}$ is the only means of specifying elementary state changes, since in $\mathcal{T}_\mathcal{R}$ there is no built-in semantics for elementary updates. Elementary updates are treated as a parameter of the logic. The rationale behind this approach is that changes to logical theories belong to several classes with distinct semantics. For example, researchers distinguish between change recording updates and revisions (e.g., [Win88, KM90]). One can therefore distinguish several update operators. $\mathcal{T}_\mathcal{R}$ can easily be augmented with new update operators by inserting appropriate axioms into the transition base, with no modifications to the model or the proof theory of the logic. In effect, each set of elementary transitions gives rise to a different version of the logic.

A transaction is executed with respect to some particular transaction base, **P**, and transition base, $\mathcal{B}$, and queries or updates a database, **D**. If the transaction updates **D**, then **D** changes state every time an elementary update is invoked. If the transaction is merely a query, then **D** remains intact.

3.2.2 Syntax

The alphabet of a language of $\mathcal{T}_\mathcal{R}$ consists of the following symbols:

- Logical symbols
  - A countably-infinite set of variables $\mathcal{V}$.
  - Predefined propositional constants: `state`, `arc`, `path`.
  - Logical connectives: $\wedge$, $\vee$ (classical conjunction and disjunction), $\otimes$, $\oplus$ (serial conjunction and disjunction), and $\neg$ (classical negation).
  - Auxiliary symbols: "(", ")", "$\langle$", "$\rangle$", and ",".
- Parameters

- A countable set, $\mathcal{P}$, of predicate symbols. For each positive integer n, $\mathcal{P}$ has a subset of symbols (possibly empty), called n-place predicate symbols.
- A set of function symbols, $\mathcal{F}$. For each positive integer n, $\mathcal{F}$ has a subset of symbols (possibly empty), called n-place function symbols. Constants are 0-place function symbols.
- Quantifiers: $\forall$, $\exists$.

Terms are defined as usual in first-order logic: A variable or constant is a term; if f is an n-place function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Formulae in $\mathcal{T}_{\mathcal{R}}$ are either *transaction formulas* or *elementary state transitions*, defined as follows.

Definition 3.1 (*Transaction formulae)* An *atomic* transaction formula (or *atom*) is an expression of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms. The predefined constants `state`, `arc` and `path` are atoms. If ϕ and ψ are transaction formulae, then so are the following expressions:

$$\phi \wedge \psi,\ \ \phi \vee \psi,\ \ \phi \oplus \psi,\ \ \phi \otimes \psi,\ \ \neg\phi,\ \ (\forall X)\phi \text{ and } (\exists X)\phi,\ \text{ where } X \in \mathcal{V}.$$

$\square$

Definition 3.2 (*Elementary state transitions)* An *elementary state transition* is an expression of the form $\langle \phi, \psi \rangle u$, where ϕ and ψ are closed first-order formulae and u is a ground atomic formula. $\square$

Elementary state transitions (also referred to as *elementary transitions*) have a purely declarative meaning, since they are logical axioms. Note that they are atomic, i.e, they cannot be decomposed. The formula $\langle \phi, \psi \rangle u$ states that u is an elementary update that transforms database ϕ into database ψ. The set of elementary state transitions for u define a binary relation between databases. u is a *deterministic* update if and only if that relation is a mapping.

3.3 Horn–$\mathcal{T}_\mathcal{R}$ Programs

Transaction Logic has a subset that is analogous to classical Horn logic. This fragment of the logic allows users to *program* transactions by using the same tool classical Horn logic uses, namely rule definitions.

A *serial Horn rule* is a formula of the form $\alpha \leftarrow \beta$, where α is an atom other than `state`, `arc` and `path`, and β is a serial conjunction. A *serial conjunction* has the form $\alpha_1 \otimes \ldots \otimes \alpha_n$, where each α_i is an atomic formula. For example, the rule $a(X,Y) \leftarrow b(X) \otimes c(Y)$ is a serial Horn rule. The $\otimes$ operator specifies that the evaluation of $b(X)$ must precede the evaluation of $c(Y)$.

From a logical point of view, the semantics of the "$\leftarrow$" connective is the classical one: $\alpha \leftarrow \beta$ is equivalent to $\alpha \vee \neg\beta$. Analogously to classical Horn logic and from a procedural viewpoint, a serial Horn rule is considered as a procedure definition, where α is the name of the procedure and β is its body. Informally, a rule $\alpha \leftarrow \beta$ can be understood in two ways, in correspondence to the two different proof-theoretic semantics it can be assigned. The first one, adopted by top-down evaluation procedures, is: "In order to show that α is true on an execution path (in order to execute α), show that β is true on the same path (execute β)." The second is a bottom-up interpretation: "If β is true on an execution path (β can be executed) then α is true on the same path (α can be executed)."

Horn–$\mathcal{T}_\mathcal{R}$ programs (or *Serial Horn programs*) are defined as follows.

Definition 3.3 (*Horn–$\mathcal{T}_\mathcal{R}$ programs*) A Horn–$\mathcal{T}_\mathcal{R}$ program is a $\mathcal{T}_\mathcal{R}$ program $(\mathcal{B}, \mathbf{P}, \mathbf{D})$, which satisfies the following conditions:

- The transaction base, $\mathbf{P}$, is a set of serial Horn rules.
- The database part, $\mathbf{D}$, must be a set of classical Horn rules. This must be true both of the current database and of every database mentioned in the transition base.
- The database $\mathbf{D}$ must be *independent* of the transaction base $\mathbf{P}$. This means that the database must not define predicates in terms of transactions, i.e., the predicate

symbols that occur in rule-heads in **P** cannot occur in rule-bodies in **D**. This must be true both of the current database and of every database mentioned in the transition base.

□

$\mathcal{T}_\mathcal{R}$ has a convenience operator, denoted & and called the *shuffle* operator, which allows for the order of goals to be unspecified. In particular,

$$\alpha \ \& \ \beta \equiv (\alpha \otimes \beta) \vee (\beta \otimes \alpha).$$

In general, a transaction, α_1, $\& \ldots \&$ α_n succeeds if and only if *some* ordering of $\alpha_1, \ldots, \alpha_n$ succeeds. Thus, every rule with serial conjunctions and shuffle operators can be rewritten as a semantically equivalent set of serial Horn rules. From now on, we allow the shuffle operator to be present in Horn–$\mathcal{T}_\mathcal{R}$ programs.

Furthermore, we use the terms "clause" and "rule" to refer to both classical and serial Horn clauses, and adopt the terms "definite clause", "negative clause" and "empty clause" (defined in Section 2.1) for Horn–$\mathcal{T}_\mathcal{R}$. We also refer to Horn–$\mathcal{T}_\mathcal{R}$ programs consisting entirely of definite clauses as *definite $\mathcal{T}_\mathcal{R}$ programs*, or simply as *definite programs*.

3.4 Model Theory

The most basic notions in $\mathcal{T}_\mathcal{R}$ are those of *state* and *execution path*. A state is defined to be a *set* of first-order semantic structures. Each state s corresponds to a specific database (read "stored facts"), namely the database having precisely those models that comprise s. The execution of a transaction may cause the database **D** to change, passing through any number of intermediate states to a final state. In $\mathcal{T}_\mathcal{R}$, any finite sequence of states is called a *path*. A path $<s_1, s_2, \ldots, s_n>$, $n \geq 2$, corresponds to a transaction that updates the database; a path $<s>$ corresponds to a non-updating transaction, i.e.

a query. [1]

Every transaction execution is therefore associated with some path. Analogously to query evaluation in FOL, from a logical perspective transaction execution corresponds to the notion of formula satisfaction. Since every transaction execution corresponds to a path, *evaluation of formulae is made on paths*, rather than on database states. Because of the emphasis on paths, semantic structures in $\mathcal{T}_{\mathcal{R}}$ are called *path structures*.

3.4.1 Path Structures and Satisfaction

Path structures are a straightforward extension of first-order structures: They assign meaning to the parameters of the logic, and form the basis for logical satisfaction. Path structures consist of a set of first-order structures, which share the same domain, U, and the same interpretation of function symbols, $I_{\mathcal{F}}$. Each state of a database, $\mathbf{D}$, is a non-empty set of such structures. The assignment of truth to atomic transaction formulae on paths is achieved through a mapping, I_{path}, that assigns to each path, π, a first-order semantic structure, $I_{path}(\pi)$. $I_{path}(\pi)$ specifies all the atoms that are true on π, and it is independent of the subpaths of π.

Definition 3.4 (*Path structures)* Let $\mathcal{L}$ be a language of $\mathcal{T}_{\mathcal{R}}$ with the set of function symbols $\mathcal{F}$. A *path structure*, $\mathbf{M}$, over $\mathcal{L}$ is a quadruple $\langle U, I_{\mathcal{F}}, N, I_{path}\rangle$, such that

- U is a set, called the domain of $\mathbf{M}$.
- $I_{\mathcal{F}}$ is an interpretation of function symbols in $\mathcal{L}$. It assigns a function $U^n \leftarrow U$ to every n-place function symbol $\mathcal{F}$.

  Given U and $I_{\mathcal{F}}$, let $Struct(U, I_{\mathcal{F}})$ denote the set of all classical first-order semantic structures over $\mathcal{L}$ of the form $\langle U, I_{\mathcal{F}}, I_{\mathcal{P}}\rangle$.
- N is a non-empty set of states, where each state is a non-empty subset of $Struct(U, I_{\mathcal{F}})$. An element of N is called a state of the path structure.

[1] The difference between s and $<s>$ is that the former represents stored facts, whereas the latter represents *derived* facts, as well.

A *path* of length k in N is any finite sequence of states, $<s_1, \ldots, s_k>$, where we define $first(\pi) = <s_1>$ and $last(\pi) = <s_k>$. The set of all paths in N is denoted by $Paths(N)$.

- I_{path} is a mapping that assigns to every path in $Paths(N)$ a semantic structure in $Struct(U, I_{\mathcal{F}})$. In addition, the mapping I_{path} is subject to the restriction:

$$I_{path}(<s>) \in s, \quad \text{for every state } s.$$

This ensures that the semantic structure assigned to a path of length one is a model of the stored data.

□

A *split* of a path $\pi = <s_1, \ldots, s_n>$ is defined to be any pair of subpaths π_1 and π_2, such that $\pi_1 = <s_1, \ldots, s_i>$ and $\pi_2 = <s_i, \ldots, s_n>$, for some i, $1 \leq i \leq n$. We write $\pi = \pi_1 \circ \pi_2$.

Definition 3.5 (*Satisfaction*) Let $\mathbf{M} = \langle U, I_{\mathcal{F}}, N, I_{path} \rangle$ be a path structure, π an arbitrary path in N, and ν a variable assignment (defined as in FOL). Then,

1. $\mathbf{M}, \pi \models_\nu p(t_1, \ldots, t_n)$ if and only if $I_{path}(\pi) \models^c_\nu p(t_1, \ldots, t_n)$, for any atom $p(t_1, \ldots, t_n)$, where $\models^c_\nu$ denotes classical satisfaction in FOL.
2. $\mathbf{M}, \pi \models_\nu \neg\phi$ if and only if it is not the case that $\mathbf{M}, \pi \models_\nu \phi$
3. $\mathbf{M}, \pi \models_\nu \phi \vee \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ or $\mathbf{M}, \pi \models_\nu \psi$.
4. $\mathbf{M}, \pi \models_\nu \phi \wedge \psi$ if and only if $\mathbf{M}, \pi \models_\nu \phi$ and $\mathbf{M}, \pi \models_\nu \psi$.
5. $\mathbf{M}, \pi \models_\nu \phi \otimes \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ and $\mathbf{M}, \pi_2 \models_\nu \psi$ for *some* split $\pi_1 \circ \pi_2$ of π.
6. $\mathbf{M}, \pi \models_\nu \phi \oplus \psi$ if and only if $\mathbf{M}, \pi_1 \models_\nu \phi$ or $\mathbf{M}, \pi_2 \models_\nu \psi$ for *every* split $\pi_1 \circ \pi_2$ of π.
7. $\mathbf{M}, \pi \models_\nu (\exists X)\phi$ if and only if $\mathbf{M}, \pi \models_\mu \phi$ for *some* variable assignment μ that agrees with ν everywhere except on X.

8. $\mathbf{M}, \pi \models_{\nu} (\forall X)\phi$ if and only if $\mathbf{M}, \pi \models_{\mu} \phi$ for *every* variable assignment μ that agrees with ν everywhere except on X.

□

Note that truth for the classical connectives ($\wedge$, $\vee$, $\neg$) on a path depends on the path itself. Truth for the non-classical connectives ($\otimes$, $\oplus$) on a path, however, depends on the subpaths of that path. Note also that if $\alpha_1, \ldots, \alpha_n$ have no effects on the database, then the expression $\alpha_1 \otimes \ldots \otimes \alpha_n$ is equivalent to $\alpha_1 \wedge \ldots \wedge \alpha_n$.

The propositional constant `state` is defined to be true on a path if and only if the path is of length one. Similarly, `arc` and `path` are defined to be true on paths of length two and on all paths, respectively. By definition,

$$\texttt{arc} \equiv (\texttt{state} \oplus \texttt{state}) \wedge \neg\texttt{state}$$

$$\texttt{path} \equiv \texttt{state} \vee \neg\texttt{state}$$

3.4.2 Models

Clearly, one is interested in path structures that are *models* of logical formulae. A model, $\mathbf{M}$, of a formula, ϕ, must satisfy ϕ on every path in that model.

Every model of a transition base must contain certain states, namely those that correspond to the databases that appear in the elementary transitions, and have a certain amount of structure. For example, if $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ appears in $\mathcal{B}$, then every model, $\mathbf{M}$, of $\mathcal{B}$ should contain at least the two states s_1 and s_2 that correspond to $\mathbf{D}_1$ and $\mathbf{D}_2$, and u should be true in $\mathbf{M}$ on the path $<s_1, s_2>$. Thus, elementary transitions impose *constraints* on paths of length two.

Transaction formulae impose constraints on *all* paths and create *dependencies* between the semantic structures on a path and its subpaths. For example, if $\alpha \leftarrow \beta \otimes \gamma$, where β and γ are ground atoms, and π is a path in a model of this formula, then the following dependency is implicit for some split, $\pi_1 \circ \pi_2$, of π.

$$\text{If} \quad I_{path}(\pi_1) \models^c \beta \quad \text{and} \quad I_{path}(\pi_2) \models^c \gamma \quad \text{then} \quad I_{path}(\pi) \models^c \alpha$$

Thus, the mapping I_{path} is not arbitrary, but satisfies constraints and dependencies that are implicit formulae. The following definitions formalize the above ideas.

Definition 3.6 (*Correspondence*) Let $\mathbf{M} = <U, \mathcal{F}, N, I_{path}>$ be a path structure. For each first-order formula $\mathbf{D}$, the expression $\mathbf{D} \rightsquigarrow s$ means "s is the set of *all* models of $\mathbf{D}$ in $Struct(U, I_{\mathcal{F}})$". We then say that s *corresponds* to the database $\mathbf{D}$. $\Box$

Definition 3.7 (*Models of Transition Bases*) Let $\mathbf{M} = <U, \mathcal{F}, N, I_{path}>$ be a path structure, let $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle$ be an elementary transition, and suppose that $\mathbf{D}_1 \rightsquigarrow s_1$ and $\mathbf{D}_2 \rightsquigarrow s_2$. Then, the elementary transition is *satisfied* in $\mathbf{M}$, denoted $\mathbf{M} \models \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$, if and only if s_1 and s_2 are states in N and $\mathbf{M}, <s_1, s_2> \models u$. $\mathbf{M}$ is said to be a *model* of a transition base, $\mathcal{B}$, if and only if $\mathbf{M}$ satisfies every transition in $\mathcal{B}$. $\Box$

Definition 3.8 (*Models of Transaction Formulae*) A path structure, $\mathbf{M}$, is a *model* of a transaction, ϕ, denoted $\mathbf{M} \models \phi$, if and only if $\mathbf{M}$, $\pi \models \phi$, for every path π in $\mathbf{M}$. A path structure is a model of a set of formulae if and only if it is a model of every formula in the set. $\Box$

3.5 Execution in $\mathcal{T}_\mathcal{R}$

3.5.1 Definition of Execution in $\mathcal{T}_\mathcal{R}$

Transaction execution in Horn–$\mathcal{T}_\mathcal{R}$ is an extension of query evaluation in classical Horn logic. Given a definite $\mathcal{T}_\mathcal{R}$-program P, a transaction α can be executed successfully if and only if α is satisfied by *every* model of P on some path, π, which starts from the *initial* database state. Thus, the difference between execution in Horn logic and execution in Horn–$\mathcal{T}_\mathcal{R}$ relies on the two different notions of satisfiability. Given a definite $\mathcal{T}_\mathcal{R}$-program P and a transaction represented as a negative clause, $\leftarrow \alpha$, we need a proof of the unsatisfiability of $\leftarrow \alpha$ on some path that starts from the initial database state. The proof procedure must find such a path and return a set of variable substitutions *associated with that path*. The situation is similar to Horn logic for queries, but it is more complex for transactions with side effects. A transaction may execute from an initial database in several ways, each corresponding to a different path. Only *one* execution is chosen, however, among the possible ones, meaning that only one database transformation is performed on the initial database. The answer to the transaction, i.e., the variable substitutions, is given with respect to the chosen execution path. Furthermore, since a transaction may query any database state on its path of execution, the variable substitutions do not depend only on the initial database state.

Transaction execution in $\mathcal{T}_{\mathcal{R}}$ is defined via the notion of *executional entailment* [BK93].

Definition 3.9 (*Executional Entailment*)
Let $\mathcal{B}$ be a transition base and $\mathbf{P}$ be a transaction base. Let α be a closed transaction formula, and let $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n$ be a sequence of databases (first-order formulae). Then, the following statement

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \alpha \tag{3.1}$$

is true if and only if, for every model, $\mathbf{M}$, of $\mathcal{B}$ and $\mathbf{P}$, there is a path $<s_0, s_1, \ldots, s_n>$ in $\mathbf{M}$ such that

- $\mathbf{M}, <s_0, s_1, \ldots, s_n> \models \alpha$; and
- $\mathbf{D}_i \rightsquigarrow s_i$, for $0 \leq i \leq n$.

□

If $P = (\mathcal{B}, \mathbf{P}, \mathbf{D}_0)$ is a Horn–$\mathcal{T}_{\mathcal{R}}$-program and $\leftarrow \alpha$ is a transaction, then α can be executed successfully from $\mathbf{D}_0$ if and only if there is a database sequence, $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_m$, such that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_m \models \exists\alpha.$$

This database sequence specifies a *possible* execution. A *correct answer substitution* for $\leftarrow \alpha$, with respect to this sequence, is a substitution θ of the variables in α, such that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_m \models \alpha\theta.$$

The database sequence $\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_m$ together with corresponding answer substitutions, specify a *correct path-answer* for α. We write path-answers as pairs, (Θ, Δ), where Θ is a set of substitutions and Δ is a database sequence. We also refer to (θ, Δ)-pairs as *partial path-answers*, where θ is a substitution associated with Δ. We call a path-answer, (Θ, Δ), *complete* if Θ consists of precisely those substitutions, θ, such that (θ, Δ) is a correct partial path-answer for α.

Example 3.1 Consider a Horn–$\mathcal{T}_\mathcal{R}$ program $(\mathcal{B}, \mathbf{P}, \mathbf{D})$, where $\mathbf{D}$ contains the base relations *EDGE* (corresponding to the predicate *edge*) and *ROUTE* (corresponding to the predicate *rt*), $\mathbf{P}$ contains the rules

$$(r1) \qquad ireach(X, Y) \leftarrow edge(X, Z) \otimes ins{:}rt(X, Z) \otimes ireach(Z, Y).$$

$$(r2) \qquad ireach(X, Y) \leftarrow edge(X, Y) \otimes ins{:}rt(X, Y).$$

and $ins{:}rt$ is an elementary update predicate. The update $ins{:}rt(i, j)$ inserts into the base relation *ROUTE* the tuple (i, j). Thus, the transition base, $\mathcal{B}$, contains infinitely many formulae, including the following:

$$\langle\{edge(1,2), edge(1,3), edge(2,4), edge(3,4)\},$$
$$\{edge(1,2), edge(1,3), edge(2,4), edge(3,4), rt(1,2)\}\rangle ins{:}rt(1,2)$$

$$\langle\{edge(1,2), edge(1,3), edge(2,4), edge(3,4), rt(1,2)\},$$
$$\{edge(1,2), edge(1,3), edge(2,4), edge(3,4), rt(1,2), rt(2,3)\}\rangle ins{:}rt(2,3)$$

Assume, also, that the *EDGE* relation consists of the tuples $(1,2), (1,3), (2,4), (3,4)$ in the current database, $\mathbf{D}$. These tuples can be thought of a constructing a *data graph* for the relation *EDGE*.

Consider now the transaction $? - ireach(1, X)$. Intuitively, this transaction returns some vertex X in the data graph for the relation *EDGE*, which is reachable from 1, and inserts those edges that correspond to the route from 1 to X into the base relation *ROUTE*. There are, therefore, four possible executions of the given transaction. Each of them is associated with a singleton set of answers and with one execution path. The path-answers are as follows.

$a1.$ $(\{2\},\ \mathbf{D}, \mathbf{D} \cup \{rt(1,2)\})$

$a2.$ $(\{3\},\ \mathbf{D}, \mathbf{D} \cup \{rt(1,3)\})$

$a3.$ $(\{4\},\ \mathbf{D}, \mathbf{D} \cup \{rt(1,2)\}, \mathbf{D} \cup \{rt(1,2), rt(2,4)\})$

$a4.$ $(\{4\},\ \mathbf{D}, \mathbf{D} \cup \{rt(1,3)\}, \mathbf{D} \cup \{rt(1,3), rt(3,4)\})$

Note that the last two executions return the same set of answers, but correspond to different database sequences. This is because there are two ways to reach 4 from 1 in the graph that represents the *EDGE* relation. □

3.6 Executional Requirements

Soundness and Completeness

We call an evaluation method for Horn–$\mathcal{T}_\mathcal{R}$ *sound*, if the set of returned answer substitutions, Θ, together with the database sequence, Δ, of an execution compose a *correct* path-answer, (Θ, Δ). We call an evaluation method for Horn–$\mathcal{T}_\mathcal{R}$ *complete*, if it returns a *complete* path-answer, whenever one exists. We are interested in evaluation methods that are both sound and complete.

Termination

A query may have infinite path-answers, infinitely many path-answers, or both. A path-answer, (Θ, Δ), is *infinite* if and only if Θ is infinite. As in classical Horn logic, function symbols or infinite relations can lead to infinite path-answers. An infinite number of path-answers results from infinitely many database *transformations*. It is possible that the number of databases generated by the elementary updates in a program is finite, but their composition generates an infinite number of path-answers.

A natural requirement would be that an evaluation method for Horn–$\mathcal{T}_\mathcal{R}$ *terminate whenever there is a finite path-answer*. This gives rise to the issue of transaction *safety*. In classical Horn logic, a query is said to be safe if it returns a finite number of answers. Extending this definition, we call a Horn–$\mathcal{T}_\mathcal{R}$ transaction *safe* if and only if it has a finite path-answer. Unfortunately, safety in Horn logic is undecidable. Since Horn–$\mathcal{T}_\mathcal{R}$ is an extension of Horn logic, this result applies to Horn–$\mathcal{T}_\mathcal{R}$, as well. Thus, we cannot always determine if there is a finite path-answer.

Example 3.2 Consider the rules

$$q(X) \leftarrow p(X).$$
$$p(f(X)) \leftarrow p(X).$$
$$p(1).$$

and the query $? - q(X)$. Clearly, this query is not safe; the set of answers are all the atoms of the form $q(f(\ldots f(1)\ldots))$. Although bottom-up evaluation enumerates all the answers, a top-down method gets stuck in a loop, trying to prove the goals $q(X_0), p(X_0), p(X_1), \ldots$. Thus, it does not return any answers, at all.

Assume, now, that the first rule is replaced by

$$q(X) \leftarrow p(X) \ \& \ b(x).$$

where b(X) is a finite relation. Then the given query is safe, since the subgoal $b(X)$ restricts the relation for $p(X)$ to a finite set of values. Thus, a clever top-down interpreter which computes the subgoal $b(X)$ first terminates, in contrast to bottom-up computation which produces the totality of the least fixpoint. □

Example 3.3 Consider the program

$$(r1) \quad a(X) \leftarrow b(X).$$
$$(r2) \quad a(X) \leftarrow ins{:}c \otimes d(X).$$

$(r3) \quad d(X) \leftarrow del{:}c \otimes a(X).$

$\langle \mathbf{D}_0, \mathbf{D}_1 \rangle ins{:}c$

$\langle \mathbf{D}_1, \mathbf{D}_0 \rangle del{:}c$

$\ldots$

with initial database $\mathbf{D}_0$. The elementary update $ins{:}c$ inserts into the database the tuple c, and $del{:}c$ deletes it. Assume that $b(X)$ is a query that is true on $\mathbf{D}_0$. Then, execution of $a(X)$ performs a number of consecutive insertions and deletions of c, and terminates by executing the query $b(X)$. Thus, there is an infinite number of possible executions of $a(X)$, corresponding to the path-answers of the form $(x,\ \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_0, \ldots, \mathbf{D}_1, \mathbf{D}_0)$, Note that two databases suffice to cause the generation of an infinite number of paths.

Assume, now, that $b(X)$ is false on $\mathbf{D}_0$. Then, $? - a(X)$ leads to failure, since there is no finite path on which $a(X)$ is true.

□

Efficiency

Logical consequences that contribute to the answer for a given query are referred to as *relevant*. All other (irrelevant) facts are considered redundant. Redundancy in Horn–$\mathcal{T}_{\mathcal{R}}$ may appear in two forms, namely as irrelevant facts and as irrelevant paths (as opposed to classical Horn logic, which exhibits only the former kind of redundancy). The composition of irrelevant paths contributes to fact redundancy, by causing irrelevant logical consequences to be generated on those paths. Thus, a key issue in transaction execution, which does not arise in query evaluation, is the composition and maintenance of relevant paths. Given a definite program and a transaction to be executed, the set of possible execution paths is a small fraction of those paths that start from the initial database, and from those only one path corresponds to the actual execution of the transaction. The situation is depicted in Figure 3.1.

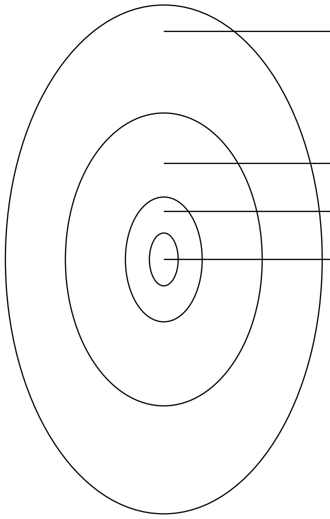

Figure 3.1: Hierarchical characterization of facts produced by a computational method.

The generation of paths and facts is made through *derivations*, i.e., sequences of inferences, by using the rules in the forward or backward direction. There is a one-to-many relationship between paths and derivations, i.e., several derivations may correspond to the same path. For example, assume that the transaction α is defined through the rules

$$(r1) \qquad \alpha(X) \leftarrow q_1(X) \otimes v.$$

$$(r2) \qquad \alpha(X) \leftarrow q_2(X) \otimes v.$$

where v is an elementary update, and q_1, q_2 are EDB predicates. If α is executed on $\mathbf{D}_0$, and $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle v$ is in the transition base, then the path that corresponds to $\mathbf{D}_0, \mathbf{D}_1$ has two derivations, one for each clause for α, and the answer substitution is the *union* of the answer substitutions for $q_1(X)$ and $q_2(X)$. Thus, these two clauses are equivalent to the following rule:

$$(r) \qquad \alpha(X) \leftarrow [q_1(X) \vee q_2(X)] \otimes v.$$

Execution can be divided in two primary tasks: Finding a path, and finding all answer substitutions on that path. The process of finding a path may involve some "redundant" path construction, due to derivations that lead to failure. The search for answer substitutions on the chosen path is carried out by following alternative derivations and comparing the corresponding paths with the chosen path. Thus, when searching for answer substitutions on the chosen execution path, inevitably some other paths are partially constructed and subsequently rejected. In effect, redundancy is a vague notion. What should be avoided is the parallel construction of several path-answers. Since only one is needed, it is naive to construct several of them, incurring the high cost of multiple database maintenance. Therefore, an efficient computational method for transaction execution should be able to *focus on one path-answer*.

Optimization techniques are desirable to facilitate choosing a path-answer with the minimum computational cost and with the minimum probability of failure. Furthermore, additional control is necessary to guide navigation in the search space when seeking answer substitutions on a chosen path. The control should detect and avoid derivations that do not construct the chosen path, and therefore do not contribute any answer substitutions to the path-answer.

As a final comment, note that we do *not* require Horn–$\mathcal{T}_{\mathcal{R}}$ interpreters to behave as random path-answer generators. That is, we do not expect a different path-answer to be generated every time the transaction is executed on a specific initial database state. We only state that the choice of path-answer is made freely, as long as it is correct. Hence, a control and optimization strategy may select the same path-answer every time the transaction is executed at the same state.

Chapter 4

Semantics of Definite $\mathcal{T}_{\mathcal{R}}$ Programs

Horn–$\mathcal{T}_{\mathcal{R}}$ programs, being declarative in nature, only state properties of their intended meaning. There are alternative ways to interpret a Horn–$\mathcal{T}_{\mathcal{R}}$ program. In this chapter we discuss program semantics from two different, though interrelated, viewpoints, namely a model-theoretic and a proof-theoretic viewpoint. In Section 4.1 we define the model-theoretic semantics of definite Horn–$\mathcal{T}_{\mathcal{R}}$ programs in analogy to the definition of the model-theoretic semantics of classical definite programs [Llo87]. In Section 4.2 we define the proof-theoretic semantics of definite programs through fixpoints, and we prove it to be equivalent to the model-theoretic semantics. Finally, in Section 4.3 we require that any interpreter of Horn–$\mathcal{T}_{\mathcal{R}}$ conform with the model-theoretic and proof-theoretic semantics.

4.1 Model-Theoretic Interpretation

From a model-theoretic viewpoint the rules in a definite $\mathcal{T}_{\mathcal{R}}$ program, P, define possible worlds, represented as path structures which are models of P. Each model of P represents a possible interpretation of the function symbols and the formulae in P. One would like to consider a uniform interpretation of function symbols throughout all $\mathcal{T}_{\mathcal{R}}$ programs. Furthermore, for a given program P, only those states that correspond to the databases mentioned in the transition base are of interest. As in classical Horn logic, where one focuses on Herbrand interpretations, we shall consider only path structures in a language, $\mathcal{L}$, of $\mathcal{T_R}$ that are *Herbrand path structures*, i.e., structures with the following properties.

1. The domain of a Herbrand path structure is the *Herbrand universe*, $U_{\mathcal{L}}$, of $\mathcal{L}$. The Herbrand universe for $\mathcal{L}$ is the set of all ground terms, which can be formed out of the constants and function symbols appearing in $\mathcal{L}$.

2. If f is an n-place function symbol in $\mathcal{L}$, then f is assigned the mapping $t_1, \ldots, t_n \rightarrow f(t_1, \ldots, t_n)$ from $(U_{\mathcal{L}})^n$ to $U_{\mathcal{L}}$. In particular, constants in $\mathcal{L}$ are assigned themselves in $U_{\mathcal{L}}$.

3. The set of states of each structure for a $\mathcal{T_R}$ program, P, consists of all and only the states that correspond to the databases mentioned in the transition base of P.

Since, for Herbrand path structures, the assignment of function symbols is fixed, it is possible to identify a first-order semantic structure in a Herbrand path structure by a set of ground atomic formulae. We thus focus on the interpretation of the formulae in P.

Example 4.1 Consider a definite $\mathcal{T_R}$ program, P, with the following transaction base:

$$(r1) \quad ireach(X, Y) \leftarrow edge(X, Z) \otimes ireach(Z, Y).$$

$$(r2) \quad ireach(X, Y) \leftarrow edge(X, Y) \otimes ins{:}arc(X, Y).$$

$ins{:}arc$ is defined in the transition base through a possibly infinite sequence of axioms of the form

$$\langle \mathbf{D}, \mathbf{D} \cup \{arc(i, j)\} \rangle ins{:}arc(i, j). \tag{4.1}$$

Suppose that the unit clauses $edge(1, 2)$ and $edge(2, 3)$ are included in some database, $\mathbf{D}$, mentioned in the transition base. Then, *every* model, $\mathbf{M}$, of P satisfies:

$$edge(1, 2) \text{ is true on the path } \mathbf{D} \text{ and} \tag{4.2}$$

$edge(2,3)$ is true on the path $\mathbf{D}$.

By Statement 4.1, $\mathbf{M}$ also satisfies:

$ins{:}arc(1,2)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(1,2)\}$ and

$ins{:}arc(2,3)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(2,3)\}$.

Combining these two facts in rule $r2$, we have that

$ireach(1,2)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(1,2)\}$,

$ireach(2,3)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(2,3)\}$. (4.3)

Combining Statements 4.2 and 4.3 in rule $r1$, we also have that

$ireach(1,3)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(2,3)\}$.

Assume now that $\mathbf{M}$ also satisfies:

$ireach(2,4)$ and $ireach(1,4)$ are true on the path $\mathbf{D}$,

$ireach(2,4)$ and $ireach(1,4)$ are true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(1,4)\}$ and

$ins{:}arc(1,2)$ is true on the path $\mathbf{D}, \mathbf{D} \cup \{arc(1,2), arc(2,1)\}$.

Then, rules $r1$ and $r2$ are still satisfied. If $\mathbf{M}$ does not violate other formulae in P, then one could accept it as a definition of the meaning of P. Intuitively, however, the satisfaction of the above facts seems to be arbitrary. They can be removed without violating the given rules. On the contrary, all previous facts cannot be removed, since they are *logical consequences* of P, that is, they *can be proven* using the rules and the elementary transitions in P. In this sense, they comprise a *minimal* interpretation of the given formulae in P. $\square$

The question of which model is the *intended interpretation* of P naturally arises. As shown by Proposition 4.2 below, the *intersection* of all Herbrand models of P is a model. This property is analogous to the model intersection property for classical Horn programs [Llo87]. We prove that this model consists precisely of those facts that can be proven using the rules and elementary transitions in P. We define this model to be the intended model of P, denoted $\mathbf{M}_P$, and we call it the *least Herbrand model* of P.

Definition 4.1 (*Intersection and union of Herbrand path structures)* Let P be a $\mathcal{T}_{\mathcal{R}}$ program in a language, $\mathcal{L}$, of $\mathcal{T}_{\mathcal{R}}$, and $\mathcal{M}_P$ be a set of Herbrand path structures for P. The *intersection* of all the path structures $\mathbf{M}_i = \langle U_{\mathcal{L}}, I_{\mathcal{F}_{\mathcal{L}}}, N_i, I_{path_i}\rangle$ in $\mathcal{M}$ is defined to be the quadruple

$$\langle U_{\mathcal{L}}, I_{\mathcal{F}_{\mathcal{L}}}, N, I_{path}\rangle = \bigcap_{\mathbf{M}_i \in \mathcal{M}} \mathbf{M}_i,$$

where I_{path} is the mapping $\pi \rightarrow \bigcap_{\mathbf{M}_i \in \mathcal{M}} I_{path_i}(\pi)$. The *union* of the structures $\mathbf{M}_i$ is defined to be the path structure

$$\langle U_{\mathcal{L}}, I_{\mathcal{F}_{\mathcal{L}}}, N, I_{path}\rangle = \bigcup_{\mathbf{M}_i \in \mathcal{M}} \mathbf{M}_i,$$

where I_{path} is the mapping $\pi \rightarrow \bigcup_{\mathbf{M}_i \in \mathcal{M}} I_{path_i}(\pi)$. $\Box$

Recall that a path structure is valid if and only if it satisfies the following two restrictions. First, its set of states is non-empty. This is because a path structure with no states is trivially a model of any program. Second, its mapping, I_{path}, satisfies the restriction $I_{path}(<s>) \in s$, for every state s. This ensures that the structure assigned to a path of length one is a model of the stored data. Furthermore, the set of states of a *Herbrand* path structure for a given program, P, contains exactly the states that correspond to the databases mentioned in the transition base of P. We show that the *intersection* of a set of path structures for a $\mathcal{T}_{\mathcal{R}}$ program, P, is a valid Herbrand path structure for P.

Proposition 4.1 Let P be a $\mathcal{T}_\mathcal{R}$ program, and let $\mathcal{M}_P$ be a set of Herbrand path structures for P. The intersection of all the path structures, $\mathbf{M}_i = \langle U_\mathcal{L}, I_{\mathcal{F}\mathcal{L}}, N_i, I_{path_i}\rangle$, in $\mathcal{M}_P$ is a valid Herbrand path structure for P.

Proof Let $\mathbf{M} = \langle U_\mathcal{L}, I_{\mathcal{F}\mathcal{L}}, N, I_{path}\rangle$ be the intersection. Clearly, N contains precisely the states that correspond to the databases mentioned in the transition base. It remains to be shown that $I_{path}(<s>) \in s$, for every $s \in N$. Let $s \in N = \bigcap_{\mathbf{M}_i \in \mathcal{M}_P} N_i$. Then, for each model $\mathbf{M}_i$, $I_{path_i}(<s>) \in s$. Since s corresponds to a database mentioned in the transition base, s is closed under first-order semantic structure intersection [Llo87]. Thus, $\bigcap_{\mathbf{M}_i \in \mathcal{M}_P} I_{path_i}(<s>) \in s$, that is, $I_{path}(<s>) \in s$. $\square$

Unfortunately, the *union* of two path structures may *not* be a valid path structure. For example, assume that a database, $\mathbf{D}$, consists of the single rule $\alpha \leftarrow \beta \ \&\ \gamma$. Then, $\{\beta\}$ and $\{\gamma\}$ are models of $\mathbf{D}$. Let s be the state that corresponds to $\mathbf{D}$, and consider two path structures with mappings I_{path_1} and I_{path_2}, such that

$$I_{path_1}(<s>) = \{\beta\} \text{ and } I_{path_2}(<s>) = \{\gamma\}.$$

Then, the mapping I_{path} of their union has the property that

$$I_{path}(<s>) = I_{path_1}(<s>) \cup I_{path_2}(<s>) = \{\beta\} \cup \{\gamma\} = \{\beta, \gamma\}.$$

However, $\{\beta, \gamma\}$ is not a model of the database, i.e., $I_{path}(<s>) \notin s$.

We prove now the model intersection property for Horn–$\mathcal{T}_\mathcal{R}$ programs.

Proposition 4.2 (*Model Intersection Property for $\mathcal{T}_\mathcal{R}$ programs*) Let P be a definite $\mathcal{T}_\mathcal{R}$ program, and let $\mathcal{M}$ be a set of Herbrand models of P. Then, the intersection of all the models in $\mathcal{M}$ is a Herbrand model of P.

Proof Let $\mathbf{M} = \langle U_\mathcal{L},\ I_{\mathcal{F}\mathcal{L}},\ N,\ I_{path}\rangle$ be the intersection $\bigcap_{\mathbf{M}_i \in \mathcal{M}} \mathbf{M}_i$. By Proposition 4.1, $\mathbf{M}$ is a Herbrand path structure. It remains to be shown that $\mathbf{M}$ is a model of P.

We distinguish the following cases:

1. Let $\phi \equiv \langle \mathbf{D}_1, \mathbf{D}_2 \rangle u$ be a ground instance of an elementary transition in the transition base of P. There exist states $s_0, s_1 \in N$, which correspond to $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively. Then, in each model M_i, $u \in I_{path_i}(<s_0, s_1>)$; hence, $u \in \bigcap_{M_i \in \mathcal{M}} I_{path_i}(<s_0, s_1>)$, that is $\mathbf{M}, <s_0, s_1> \models u$.

2. Let $\phi \equiv \alpha \leftarrow \beta$ be a ground instance of a transaction formula in the transaction base of P. Let β be $\beta_1 \otimes \ldots \otimes \beta_n$, where each β_i is atomic. We first prove the following statement, by induction on n.

$$\text{For each model } \mathbf{M}_i, \quad \mathbf{M}, \pi \models \beta \ \Rightarrow \ \mathbf{M}_i, \pi \models \beta. \tag{4.4}$$

Basis. Let $n = 1$. Then, $I_{path}(\pi) \models^c \beta_1$. By the construction of I_{path}, we have that for each I_{path_i}, $I_{path_i}(\pi) \models^c \beta_1$. Hence, for each model $\mathbf{M}_i$, $\mathbf{M}_i, \pi \models \beta_1$.

Induction step. Suppose the result holds for $n < k$. We prove that it also holds for $n = k$. Let $\mathbf{M}, \pi \models \beta_1 \otimes \ldots \otimes \beta_k$. Then, there is a split, $\pi_1 \circ \pi_2$, of π, such that $\mathbf{M}, \pi_1 \models \beta_1 \otimes \ldots \otimes \beta_{k-1}$ and $\mathbf{M}, \pi_2 \models \beta_k$. By induction hypothesis, for each $\mathbf{M}_i$, $\mathbf{M}_i, \pi_1 \models \beta_1 \otimes \ldots \otimes \beta_{k-1}$ and $\mathbf{M}_i, \pi_2 \models \beta_k$. Hence, $\mathbf{M}_i, \pi \models \beta$.

Now, $\mathbf{M}, \pi \models \phi$ if and only if $\mathbf{M}, \pi \models \beta \ \Rightarrow \ \mathbf{M}, \pi \models \alpha$. Assume that $\mathbf{M}, \pi \models \beta$. By Statement 4.4, $\mathbf{M}_i, \pi \models \beta$, for each model $\mathbf{M}_i$. Hence, $\mathbf{M}_i, \pi \models \alpha$, which implies that $\alpha \in I_{path_i}(\pi)$. Hence, $\alpha \in I_{path}(\pi)$, that is, $\mathbf{M}, \pi \models \alpha$.

$\square$

The choice of intended model is not unjustified. As shown in Theorem 4.1 below, $\mathbf{M}_P$ consists of the *logical consequences* of P. Furthermore, its set of states contains precisely those states that correspond to the databases mentioned in the transition base of P. Thus, $\mathbf{M}_P$ satisfies our intuition of what is necessarily true and what is redundant and therefore should be taken to be false with respect to the given program, P.

Definition 4.2 (*Logical consequence on paths*) Let P be a $\mathcal{T}_\mathcal{R}$ program and α a closed formula in a language, $\mathcal{L}$, of Transaction Logic. If, for *every* path structure $\mathbf{M}$ of $\mathcal{L}$,

$\mathbf{M}$ is a model of P implies $\mathbf{M}, \pi \models \alpha$ for some π in $\mathbf{M}$, then we say that α is a *logical consequence* of P on path π. $\square$

Theorem 4.1 Let $\mathcal{L}$ be a language of $\mathcal{T}_\mathcal{R}$, let P be a definite $\mathcal{T}_\mathcal{R}$ program in $\mathcal{L}$, and let $\mathbf{M}_P = \langle U_\mathcal{L}, I_{\mathcal{F}\mathcal{L}}, N, I_{path}\rangle$ be the least Herbrand model of P. Then, for each path π in $Paths(N)$, $I_{path}(\pi)$ contains exactly those atoms α that are logical consequences of P on π.

Proof Let $\pi \in Paths(N)$. Then, $\pi \in Paths(N_i)$, for each model $\mathbf{M}_i$ of P. We have also that

$$
\begin{array}{ll}
 & \alpha \text{ is a logical consequence of } P \text{ on } \pi \\
\iff & \text{for each model } \mathbf{M}_i \text{ of } P,\ \mathbf{M}_i, \pi \models \alpha \\
\iff & \text{for each model } \mathbf{M}_i \text{ of } P,\ I_{path_i}(\pi) \models^c \alpha \\
\iff & \text{for each } i,\ \alpha \in I_{path_i}(\pi) \\
\iff & \alpha \in \bigcap_i I_{path_i}(\pi) \\
\iff & \bigcap_i I_{path_i}(\pi) \models^c \alpha \\
\iff & I_{path}(\pi) \models^c \alpha.
\end{array}
$$

$\square$

4.2 Proof-Theoretic Interpretation

From a proof-theoretic point of view, the rules and elementary transitions in definite $\mathcal{T}_\mathcal{R}$ programs are axioms that are used in a proof. Like classical Horn clauses, $\mathcal{T}_\mathcal{R}$ clauses can be interpreted either bottom-up or top-down. A bottom-up interpretation of a rule $\alpha \leftarrow \beta$ is: "If β is true (β can be executed) then α is true (α can be executed)." The same rule can be interpreted top-down as: "In order to show that α is true (in order to execute α), show that β is true (execute β)." Both interpretations yield the same logical implications. They give rise, however, to different proof procedures. Bonner and Kifer

[BK93] describe two proof systems based on these two interpretations.

Fixpoint concepts cast more light on the semantics of definite $\mathcal{T}_{\mathcal{R}}$ programs, and they provide another link between the declarative and procedural semantics. In the sequel, we define the bottom-up semantics for definite $\mathcal{T}_{\mathcal{R}}$ programs through fixpoints, and we show that it is equivalent to the model-theoretic semantics. This result increases our understanding of the meaning of definite $\mathcal{T}_{\mathcal{R}}$ programs. Furthermore, it will form the basis of a bottom-up evaluation method for these programs.

We first provide some preliminaries on fixpoints. A detailed presentation and the proofs of all propositions appear in [Llo87].

Preliminaries

A relation, R, on a set, S, is a *partial order* if the following conditions are satisfied:

(a) xRx, for all $x \in S$.

(b) xRy and yRx imply $x = y$, for all $x, y \in S$.

(c) xRy and yRz imply xRz, for all $x, y, z \in S$.

A partially ordered set L is a *complete lattice* if $lub(X)$ (the *least upper bound*) and $glb(X)$ (the *greatest lower bound*) exist for every subset X of L. We say that $X \subseteq L$ is *directed* if every finite subset of X has an upper bound in X. A mapping $T : L \to L$ is *monotonic* if $T(x) \leq T(y)$, whenever $x \leq y$. It is *continuous at* X if $lub(X)$ and $lub(T(X)$ exist, and $T(lub(X)) = lub(T(X))$. We say that $a \in L$ is the *least fixpoint* of T if a is a fixpoint (i.e., $T(a) = a$) and for all fixpoints b of T, we have that $a \leq b$.

Proposition 4.3 Let L be a partially ordered set, such that the $glb(X)$ exists for every subset X of L, and let $T : L \to L$ be monotonic. Then T has a least fixpoint, $lfp(T)$. Furthermore, $lfp(T) = glb\{x : T(x) \leq x\}$. $\Box$

The *ordinal powers of* T are defined as follows.

Definition 4.3 (*Ordinal powers of T*) Let L be a partially ordered set, and $T : L \to L$ be monotonic. Let also ω denote the set of natural numbers. Then:
$T \uparrow 0 = glb(L)$.
$T \uparrow n = T(T \uparrow (n-1))$, where $n \in \omega$.
$T \uparrow \omega = lub(T \uparrow n : n \in \omega)$, provided that $lub(T \uparrow n : n \in \omega)$ exists. $\square$

Proposition 4.4 Let L be a partially ordered set, such that the $glb(X)$ exists for every subset X of L, and let $T : L \to L$ be monotonic. Then, for all $n \in \omega$, $T \uparrow n \leq lfp(T)$. $\square$

Proposition 4.5 Let L be a partially ordered set, such that the $glb(X)$ exists for every subset X of L, and let $T : L \to L$ be continuous at $\{T \uparrow n : n \in \omega\}$. Then $lfp(T) = T \uparrow \omega$. $\square$

We now wish to use fixpoint concepts for characterizing definite $\mathcal{T_R}$ programs. Given a definite $\mathcal{T_R}$ program, P, we will define a monotonic mapping on the set of all Herbrand path structures for P, and will show that the least Herbrand model of P is the least fixpoint of that mapping.

4.2.1 Fixpoint Characterization

In what follows, we use the notations $I^{\mathbf{M}}_{path}$ and $N^{\mathbf{M}}$ to denote the mapping and set of states of the structure $\mathbf{M}$, respectively.

We need to define a partial order on the set of Herbrand path structures. This is done as follows. Let $\mathcal{M}_P$ denote the set of all Herbrand path structures for a definite $\mathcal{T_R}$ program, P. We define the following ordering relation on $\mathcal{M}_P$.

Definition 4.4 (*Inclusion relation on path structures*) Let $\mathbf{M}_1 = \langle U, I_{\mathcal{F}}, N_1, I_{path_1} \rangle$ and $\mathbf{M}_2 = \langle U, I_{\mathcal{F}}, N_2, I_{path_2} \rangle$ be two Herbrand path structures, and P a definite $\mathcal{T_R}$ program. Then we say that $\mathbf{M}_1$ *is included in* $\mathbf{M}_2$, denoted $\mathbf{M}_1 \leq \mathbf{M}_2$, if and only if $I_{path_1}(\pi) \subseteq I_{path_2}(\pi)$, for each path, π, in $Paths(N_1)$. $\square$

Intuitively, $\mathbf{M}_1 \leq \mathbf{M}_2$ means that $\mathbf{M}_2$ proves at least the same facts as $\mathbf{M}_1$.

Proposition 4.6 Let $\mathbf{M}_1$ and $\mathbf{M}$ be Herbrand path structures, such that $\mathbf{M}_1 \leq \mathbf{M}_2$. Then, for each closed serial conjunction, α, and each path π in $\mathbf{M}_1$,

$$\mathbf{M}_1, \pi \models \alpha \Rightarrow \mathbf{M}_2, \pi \models \alpha.$$

Proof Straightforward induction on the length of α. $\Box$

The inclusion relation on path structures is a partial order (since set inclusion is a partial order). Let X be a subset of $\mathcal{M}_P$. As shown in Section 4.1, the union of all structures in X is not always a path structure. Therefore, the least upper bound of X does not always exist. By Proposition 4.1, the greatest lower bound of X exists, and it is the intersection of all structures in X.

The Herbrand path structure which is the intersection of all structures in $\mathcal{M}_P$ is the greatest lower bound of $\mathcal{M}_P$, denoted $\mathbf{M}_\perp$. The following proposition shows a characteristic property of $\mathbf{M}_\perp$.

Proposition 4.7 The intersection, $\mathbf{M}_\perp = \langle U_\mathcal{L}, I_{\mathcal{F}\mathcal{L}}, N_\perp, I_{path_\perp} \rangle$, of all structures in $\mathcal{M}_P$ is a path structure, with the property that $I_{path_\perp}(<s>)$ is the minimal structure in s, and $I_{path_\perp}(\pi) = \{\}$ if the length of π is greater than one.

Proof By Proposition 4.1, $\mathbf{M}_\perp$ is a Herbrand path structure. Let s be a state in $N_\perp$, which corresponds to a database, $\mathbf{D}$, mentioned in the transition base of P. By Definition 4.1,

$$I_{path_\perp}(<s>) = \bigcap_{\mathbf{M} \in \mathcal{M}_P} I_{path}{}^{\mathbf{M}}(<s>).$$

State s consists of all the models of $\mathbf{D}$. For each model $m \in s$, there is a Herbrand path structure, $\mathbf{M}$, such that $I^{\mathbf{M}}_{path}(<s>) = m$. Hence, the intersection $\bigcap_{\mathbf{M} \in \mathcal{M}_P} I_{path}{}^{\mathbf{M}}(<s>)$ is over *all* models of the database $\mathbf{D}$. By the model intersection property for classical Horn programs [Llo87], we have that $I_{path_\perp}(<s>)$ is the *least model* of $\mathbf{D}$.

Clearly, $I_{path_\perp}(\pi) = \{\}$, if the length of π is greater than one. $\Box$

We now define a mapping on $\mathcal{M}_P$, which is the link between the declarative and procedural semantics of definite $\mathcal{T}_\mathcal{R}$ programs.

Definition 4.5 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T}_\mathcal{R}$ program. We define the mapping $T_{\mathcal{B},\mathbf{P}} : \mathcal{M}_P \to \mathcal{M}_P$ as follows. Let $\mathbf{M} = \langle U_\mathcal{L}, I_{\mathcal{F}\mathcal{L}}, N, I_{path}\rangle$ be a path structure in $\mathcal{M}_P$. Then the path structure $T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) = \langle U^P, I^P_\mathcal{F}, N^P, I^P_{path}\rangle$ is constructed as follows, where all states are subsets of $Struct(U^P, I^P_\mathcal{F})$.

Set $U^P = U_\mathcal{L}$, $I^P_\mathcal{F} = I_{\mathcal{F}\mathcal{L}}$, and $N^P = N$.

The mapping I^P_{path} is constructed by the following steps.

1. For each state $s \in N$, copy $I_{path}(<s>)$ to $I^P_{path}(<s>)$.
2. For each elementary transition $\langle \mathbf{D}^1,\ \mathbf{D}^2\rangle u$ in $\mathcal{B}$, such that $\mathbf{D}^1 \leadsto s_1$ and $\mathbf{D}^2 \leadsto s_2$, insert u into $I^P_{path}(<s_1, s_2>)$.
3. For each ground instance, $\alpha \leftarrow \beta$, of a clause in $\mathbf{P}$, and each path, π, in $Paths(N^P)$, such that $\mathbf{M}, \pi \models \beta$, insert α into $I^P_{path}(\pi)$.

$\square$

The mapping $T_{\mathcal{B},\mathbf{P}}$ characterizes Herbrand path structures which are *models* of P. Specifically, for each model, $\mathbf{M}$, of the program P, $T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) \leq \mathbf{M}$.

Proposition 4.8 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a serial Horn–$\mathcal{T}_\mathcal{R}$ program and $\mathbf{M}$ a Herbrand structure in $\mathcal{M}_P$. Then, $\mathbf{M}$ is a model of P, if and only if $T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) \leq \mathbf{M}$.

Proof Let $\phi \equiv \alpha \leftarrow \beta$ be a ground instance of a clause in $\mathbf{P}$ and π be any path, or $\phi \equiv \langle \mathbf{D}_1, \mathbf{D}_2\rangle\alpha$ be a ground instance of an elementary transition in $\mathcal{B}$ and π correspond to $\mathbf{D}_1, \mathbf{D}_2$. We have that

$$\begin{aligned}
& \mathbf{M} \text{ is a model of } \phi \\
\iff & (\ \phi \equiv \alpha \leftarrow \beta \text{ and } \mathbf{M}, \pi \models \beta\ ) \ \Rightarrow\ \mathbf{M}, \pi \models \alpha \\
& \ \text{or } \phi \equiv \langle \mathbf{D}_1, \mathbf{D}_2\rangle\alpha \ \Rightarrow\ \mathbf{M}, \pi \models \alpha \\
\iff & (\ \phi \equiv \alpha \leftarrow \beta \text{ and } \mathbf{M}, \pi \models \beta\ ) \ \Rightarrow\ \alpha \in I^{\mathbf{M}}_{path}(\pi) \\
& \ \text{or } \phi \equiv \langle \mathbf{D}_1, \mathbf{D}_2\rangle\alpha \ \Rightarrow\ \alpha \in I^{\mathbf{M}}_{path}(\pi) \\
\iff & \alpha \in I^{T_{\mathcal{B},\mathbf{P}}(\mathbf{M})}_{path}(\pi) \ \Rightarrow\ \alpha \in I^{\mathbf{M}}_{path}(\pi) \ \ \text{(by Definition 4.5).}
\end{aligned}$$

$\square$

It is straightforward to show that $T_{\mathcal{B},\mathbf{P}}$ is monotonic. We now show that the ordinal powers of $T_{\mathcal{B},\mathbf{P}}$ are well defined, i.e., that least upper bound of $\{T \uparrow n : n \in \omega\}$, is a valid Herbrand path structure. Specifically, it is the union $\bigcup_{n\in\omega} T \uparrow n$. We then prove that $T_{\mathcal{B},\mathbf{P}}$ is continuous at $\{T \uparrow n : n \in \omega\}$.

Proposition 4.9 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T_R}$ program. $\{T_{\mathcal{B},\mathbf{P}} \uparrow n : n \in \omega\}$ has a least upper bound in $\mathcal{M}_P$.

Proof Let $\mathbf{M}^{lub} = \langle U_{\mathcal{L}}, I_{\mathcal{F}_{\mathcal{L}}}, N^{lub}, I^{lub}_{path}\rangle$ be the least upper bound of $\{T \uparrow n : n \in \omega\}$. N^{lub} is non-empty; specifically, for each $n \in \omega$, $N^{T\uparrow n}$ contains precisely the states that correspond to the databases mentioned in the transition base.

Let s be a state in N^{lub}. We show that $I^{lub}_{path}(<s>) \in s$. Let $\mathbf{M}_n = T \uparrow n$, for some $n \in \omega$. Note that $T_{\mathcal{B},\mathbf{P}}$ copies $I^{\mathbf{M}_n}_{path}(<s>)$ into $I^{T_{\mathcal{B},\mathbf{P}}(\mathbf{M}_n)}_{path}(<s>)$, and adds into $I^{T_{\mathcal{B},\mathbf{P}}(\mathbf{M}_n)}_{path}(<s>)$ atoms defined through rules in $\mathbf{P}$. Recall that the mapping $I_{path_\perp}$ of $\mathbf{M}_\perp$ assigns $I_{path_\perp}(<s>)$ the minimal structure in s. Thus, $I^{T_{\mathcal{B},\mathbf{P}}(\mathbf{M}_n)}_{path}(<s>)$ can be viewed as the union of two sets, $S^{\mathbf{P}}_n$ and $S^{\mathbf{D}}$, where $S^{\mathbf{D}}$ is the minimal structure in s. Consequently,

$$I^{lub}_{path}(<s>) = \bigcup_n I^{T_{\mathcal{B},\mathbf{P}}(\mathbf{M}_n)}_{path}(<s>) = \bigcup_n S^{\mathbf{P}}_n \ \cup \ S^{\mathbf{D}}.$$

This means that $I^{lub}_{path}(<s>) \in s$. $\square$

Corollary .1 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T_R}$ program. Then, the set $\{T_{\mathcal{B},\mathbf{P}} \uparrow n : n \in \omega\}$ is directed. $\square$

Claim 4.1 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T_R}$ program. Let X be $\{T_{\mathcal{B},\mathbf{P}} \uparrow n : n \in \omega\}$, and α an instance of serial conjunction. Then,

$$lub(X), \pi \models \alpha \Longleftrightarrow \mathbf{M}, \pi \models \alpha, \text{ for some } \mathbf{M} \in X.$$

Proof Let α be $\alpha_1 \otimes \ldots \otimes \alpha_n$, where each α_i is atomic. We first prove that

$$lub(X), \pi \models \alpha \;\Rightarrow\; \mathbf{M}, \pi \models \alpha, \text{ for some } \mathbf{M} \in X$$

by induction on n. Let $\pi \in Paths(N^{lub(X)})$.

Basis ($n = 1$). We have that

$$\begin{aligned}
& lub(X), \pi \models \alpha_1 \\
\Rightarrow\;& I^{lub(X)}_{path}(\pi) \models^c \alpha_1 \\
\Rightarrow\;& I^{\mathbf{M}}_{path}(\pi) \models^c \alpha_1, \text{ for some } \mathbf{M} \in X \text{ (by Definition 4.4)} \\
\Rightarrow\;& \mathbf{M}, \pi \models \alpha_1 \text{ for some } \mathbf{M} \in X.
\end{aligned}$$

Induction step Suppose the result holds for $n < k$. We prove that it also holds for $n = k$.

$$\begin{aligned}
& lub(X), \pi \models \alpha_1 \otimes \ldots \otimes \alpha_k \\
\Rightarrow\;& lub(X), \pi_1 \models \alpha_1 \otimes \ldots \otimes \alpha_{k-1} \text{ and } lub(X), \pi_2 \models \alpha_k, \text{ for some spilt } \pi_1 \circ \pi_2 \text{ of } \pi \\
\Rightarrow\;& \mathbf{M}_1, \pi_1 \models \alpha_1 \otimes \ldots \otimes \alpha_{k-1} \text{ and } \mathbf{M}_2, \pi_2 \models \alpha_k, \text{for some spilt, } \pi_1 \circ \pi_2, \text{ of } \pi, \\
& \text{and for some } \mathbf{M}_1, \mathbf{M}_2 \in X \text{ (by the induction hypothesis).}
\end{aligned}$$

Since X is directed (by Corollary .1), and $\{\mathbf{M}_1, \mathbf{M}_2\}$ is a finite subset of X, $\{\mathbf{M}_1, \mathbf{M}_2\}$ has an upper bound, $\mathbf{M}$, in X. Thus, for some $\mathbf{M} \in X$,

$$\begin{aligned}
& \mathbf{M}, \pi_1 \models \alpha_1 \otimes \ldots \otimes \alpha_{k-1} \text{ and } \mathbf{M}, \pi_2 \models \alpha_k, \text{ for some spilt, } \pi_1 \circ \pi_2, \text{ of } \pi \\
\Rightarrow\;& \mathbf{M}, \pi \models \alpha_1 \otimes \ldots \otimes \alpha_k.
\end{aligned}$$

We now prove the other direction, i.e.,

$$lub(X), \pi \models \alpha \Leftarrow \mathbf{M}, \pi \models \alpha, \text{ for some } \mathbf{M} \in X.$$

Let $\mathbf{M}$ be a model in X, such that $\mathbf{M}, \pi \models \alpha$, for each path π in $\mathbf{M}$. By Definition 4.4, we have that $\mathbf{M} \leq lub(X)$. Hence, $lub(X), \pi \models \alpha$. $\Box$

Proposition 4.10 Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T_R}$ program. Then the mapping $T_{\mathcal{B},\mathbf{P}}$ is continuous at $\{T_{\mathcal{B},\mathbf{P}} \uparrow n : n \in \omega\}$.

Proof Let $X = \{T_{\mathcal{B},\mathbf{P}} \uparrow n : n \in \omega\}$. By Proposition 4.9, X has a least upper bound. In order to prove that $T_{\mathcal{B},\mathbf{P}}$ is continuous at X, we must show that $T_{\mathcal{B},\mathbf{P}}(lub(X)) = lub(T_{\mathcal{B},\mathbf{P}}(X))$.

We have that

$$
\begin{aligned}
& \mathbf{M} \leq lub(X), \text{ for each } \mathbf{M} \in X \\
\Rightarrow\ & T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) \leq T_{\mathcal{B},\mathbf{P}}(lub(X)), \text{ for each } \mathbf{M} \in X \text{ (because } T_{\mathcal{B},\mathbf{P}} \text{ is monotonic)} \\
\Rightarrow\ & lub\{T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) : \mathbf{M} \in X\} \leq T_{\mathcal{B},\mathbf{P}}(lub(X)) \\
\Rightarrow\ & lub(T_{\mathcal{B},\mathbf{P}}(X)) \leq T_{\mathcal{B},\mathbf{P}}(lub(X)).
\end{aligned}
$$

We now show that $T_{\mathcal{B},\mathbf{P}}(lub(X)) \leq lub(T_{\mathcal{B},\mathbf{P}}(X))$. Specifically, we prove that

$$I_{path}^{T_{\mathcal{B},\mathbf{P}}(lub(X))}(\pi) \subseteq I_{path}^{lub(T_{\mathcal{B},\mathbf{P}}(X))}(\pi),$$

for each path π. Let α be ground atom in $I_{path}^{T_{\mathcal{B},\mathbf{P}}(lub(X))}(\pi)$. The, by Definition 4.5, there are three possible cases:

1. π is of the form $<s>$, and $\alpha \in I_{path}^{lub(X)}(<s>)$. We have that

$$
\begin{aligned}
& \alpha \in I_{path}^{lub(X)}(<s>) \\
\Rightarrow\ & lub(X), <s> \models \alpha \\
\Rightarrow\ & \mathbf{M}, \pi \models \alpha, \text{ for some } \mathbf{M} \in X \text{ (by Claim 4.1)} \\
\Rightarrow\ & I_{path}^{\mathbf{M}}(<s>) \models \alpha \\
\Rightarrow\ & T_{\mathcal{B},\mathbf{P}}(\mathbf{M}), \pi \models \alpha \text{ (by Definition 4.5).}
\end{aligned}
$$

2. π is of the form $<s_1, s_2>$, $\mathbf{D}_1 \rightsquigarrow s_1$, $\mathbf{D}_2 \rightsquigarrow s_2$, and $\langle \mathbf{D}_1, \mathbf{D}_2 \rangle \alpha$ is an elementary transition in the transition base. By Definition 4.5, for any $\mathbf{M} \in X$, $T_{\mathcal{B},\mathbf{P}}(\mathbf{M}), \pi \models \alpha$.

3. α is the head of a ground instance, $\alpha \leftarrow \beta$, of a rule in $\mathbf{P}$. We have that

$$\begin{aligned}
& \alpha \in I_{path}^{T_{\mathcal{B},\mathrm{P}}(lub(X))}(\pi) \\
\Rightarrow\ & T_{\mathcal{B},\mathrm{P}}(lub(X)), \pi \models \alpha. \\
\Rightarrow\ & lub(X), \pi \models \beta \\
\Rightarrow\ & \mathbf{M}, \pi \models \beta, \text{ for some } \mathbf{M} \in X \text{ (by Claim 4.1)} \\
\Rightarrow\ & T_{\mathcal{B},\mathbf{P}}(\mathbf{M}), \pi \models \alpha \text{ (by Definition 4.5).}
\end{aligned}$$

In all three cases, we have that, for some $\mathbf{M} \in X$,

$$\begin{aligned}
\Rightarrow\ & \alpha \in I_{path}^{T_{\mathcal{B},\mathrm{P}}(\mathbf{M})}(\pi) \\
\Rightarrow\ & \alpha \in I_{path}^{lub(T_{\mathcal{B},\mathrm{P}}(X))}(\pi).
\end{aligned}$$

$\square$

Since $T_{\mathcal{B},\mathbf{P}}$ is monotonic, it has a least fixpoint. It is proven that *the least fixpoint of* $T_{\mathcal{B},\mathbf{P}}$ *is the least Herbrand model of* P.

Theorem 4.2 (*Fixpoint characterization of the Least Herbrand Model)* Let P be a definite $\mathcal{T}_{\mathcal{R}}$ program. Then $\mathbf{M}_P = lfp(T_{\mathcal{B},\mathbf{P}}) = T_{\mathcal{B},\mathbf{P}} \uparrow \omega$.

Proof

$$\begin{aligned}
M_P &= glb\{\mathbf{M} : \mathbf{M} \in \mathcal{M}_P\} \\
&= glb\{\mathbf{M} : T_{\mathcal{B},\mathbf{P}}(\mathbf{M}) \leq \mathbf{M}\} \text{ (by Proposition 4.8)} \\
&= lfp(T_{\mathcal{B},\mathbf{P}}) \text{ (by Proposition 4.3)} \\
&= T_{\mathcal{B},\mathbf{P}} \uparrow \omega \text{ (by Proposition 4.5).}
\end{aligned}$$

□

The significance of Theorem 4.2 is twofold. On the one hand, it establishes the connection between the least Herbrand model, $\mathbf{M}_P$, and the least fixpoint of $T_{\mathcal{B},\mathbf{P}}$. On the other hand, it provides a conceptual iterative algorithm for computing the least fixpoint. This algorithm starts by applying $T_{\mathcal{B},\mathbf{P}}$ to $\mathbf{M}_\perp$, then to the result and so on, according to Definition 4.3. Convergence is guaranteed by Proposition 4.4. The computation proceeds from $\mathbf{M}_\perp$, through path structures that are not models of P, to the least fixpoint. The situation is depicted in Figure 4.1.

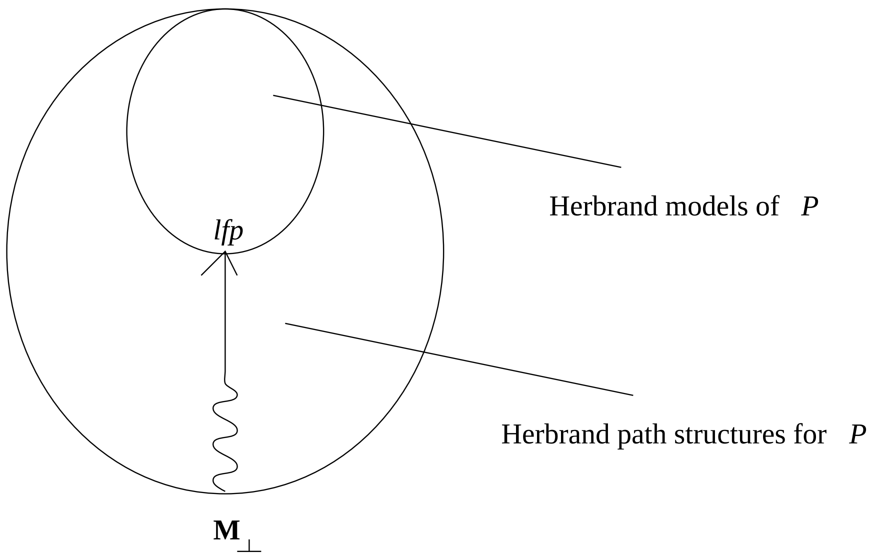

Figure 4.1: Least fixpoint computation

Example 4.2 Consider again the program P of Example 4.1. The facts produced in each step of the fixpoint computation are as follows.

1. $edge(1,2)$ and $edge(2,3)$ on $\mathbf{D}$,

   $ins{:}arc(1,2)$ on $\mathbf{D}$, $\mathbf{D} \cup \{arc(1,2)\}$,

   $ins{:}arc(2,3)$ on $\mathbf{D}$, $\mathbf{D} \cup \{arc(2,3)\}$,

   $\ldots$

2. The facts of Step 1, plus the following:

   $ireach(1,2)$ on $\mathbf{D}$, $\mathbf{D} \cup \{arc(1,2)\}$,

   $ireach(2,3)$ on $\mathbf{D}$, $\mathbf{D} \cup \{arc(2,3)\}$,

   ...

3. The facts of Step 2, plus the following:

   $ireach(1,3)$ on $\mathbf{D}$, $\mathbf{D} \cup \{arc(2,3)\}$,

   ...

4. The facts of Step 3 only (i.e., no new facts). Terminate.

$\square$

4.3 Conclusion

Another way to define the meaning of definite $\mathcal{T}_\mathcal{R}$ programs is to provide an algorithm that generates facts from the formulae in a given program P. Prolog is an example of a language which defines the meaning of rules by an algorithm that produces proofs for a given fact. Unfortunately, the set of facts for which Prolog finds a proof is not necessarily the same as the set of facts for which a proof exists.

Computational definitions of meaning are not desirable due to their often ad-hoc nature. They exist in practice only because of the unavailability of powerful enough algorithms to implement model-theoretic or proof-theoretic semantics. Therefore, we avoid to define the meaning of definite $\mathcal{T}_\mathcal{R}$ programs using computational interpretations. On the contrary, we require that any interpreter of definite $\mathcal{T}_\mathcal{R}$ programs conform with the semantics specified in the previous sections.

Chapter 5

Bottom-Up Execution of Definite $\mathcal{T_R}$ Programs

Bottom-up evaluation has been proven efficient for query evaluation in deductive databases, and it is therefore worth investigating how to extend it to handle updates. In this chapter we set up a framework and propose an approach to bottom-up evaluation of Horn–$\mathcal{T_R}$ programs. Bottom-up evaluation is based on the fixpoint semantics of Horn–$\mathcal{T_R}$. As in all logics, there is a gap, however, between the model-theory of the logic and reality. In Horn–$\mathcal{T_R}$ this concerns primarily the implementation of elementary updates, especially the execution of elementary updates, representation of execution paths, path composition, and manipulation of numerous databases. We propose solutions for these essential problems, but we do not consider low-level, i.e., implementation problems, leaving these to future research. The proposed approach is a combination of rule rewriting techniques, for adding top-down control, with bottom-up evaluation.

Section 5.1 develops a framework for bottom-up evaluation. Section 5.2 elaborates on our assumptions and requirements on implementing and executing elementary updates, and gives solutions to a number of issues related to bottom-up evaluation of Horn–$\mathcal{T_R}$ programs. Section 5.3 first introduces the reasons for which the addition of top-down functionality is mandatory both for efficiency and practical reasons. Then, it describes how top-down control is embedded into bottom-up evaluation for Horn–$\mathcal{T}_\mathcal{R}$ programs. Section 5.4 investigates the applicability of the proposed method and outlines research issues.

5.1 Setting Up the Framework for Bottom-Up Evaluation

In this section, we set up our framework for bottom-up evaluation based on the fixpoint semantics of Horn–$\mathcal{T}_\mathcal{R}$, by extending the approach taken for classical Horn logic to handle execution paths. Temporarily, we make a few unrealistic assumptions, which are dropped in Section 5.2. Under these assumptions, we describe a representation of logical consequences on paths, and a pure bottom-up evaluation method for computing the intended meaning of a Horn–$\mathcal{T}_\mathcal{R}$ program.

5.1.1 Introduction

Although the bottom-up proof-theoretic semantics of Horn–$\mathcal{T}_\mathcal{R}$ is equivalent to its model-theoretic semantics, our opinion is that it not as natural as the top-down semantics, when we program updates. Horn–$\mathcal{T}_\mathcal{R}$ has the primary characteristics of a conventional high level programming language: A naming mechanism for procedure definitions, serial execution, and, most importantly, elementary updates for the state space (i.e., database) changes. One is bound to program transactions by specifying actions to be executed in some order if certain conditions are true. Although conditions (i.e., queries) may be thought of in a bottom-up fashion, the overall approach to programming actions is clearly top-down. An implication of our intuition is that an attempt to execute $\mathcal{T}_\mathcal{R}$-programs bottom-up would resemble considerably an attempt to execute Pascal, or C programs bottom-up. From an implementation perspective, this intuition is not a severe limitation, as long as there are efficient methods for bottom-up execution.

We first show that pure bottom-up evaluation is not realistic. Using several refinements to overcome feasibility and efficiency problems, we come up with a method that adds top-down control to bottom-up evaluation. To the best of our knowledge, this is the first serious attempt at evaluating bottom-up logic programs with database updates. Although it is by no means complete, we feel that it provides significant insight to the problems of evaluating rules with updates in bodies in a bottom-up fashion.

Bottom-up evaluation of a definite program, P, computes the least model of P. Thus, for each predicate, p, and each path, π, it finds all ground instances (facts) of p that are true on π. The process of evaluation is iterative, i.e., the least model is computed in a (possibly infinite) number of steps. Essentially, each iteration simulates an application of the operator $T_{\mathcal{B},\mathbf{P}}$ (defined in Chapter 4) on known facts, which can be thought of as comprising the Herbrand path structure passed to the argument of $T_{\mathcal{B},\mathbf{P}}$.

Assumptions

In what follows, we make the following assumptions. First, we are dealing with *safe* rules [Ull89a]. This is because unsafe rules create infinite relations from finite ones. For example, a rule with a variable, X, in the head, that does not appear in the body, is not safe; X may take on an infinite set of values.

Second, for simplicity we do not deal with function symbols. Function symbols can be treated in standard ways [Ull89a, Ull89b].

Finally, we assume that for each intensional predicate, p, all rules for p are *rectified* [Ull89a]. This means that all their heads are identical, of the form $p(X_1, \ldots, X_n)$, for distinct variables $X_1, \ldots, X_n$. Any program can be rewritten to satisfy our assumption [Ull89a].

Terminology

In the rest of this thesis, we use the term "path" to refer to both sequences of states and sequences of databases. The model theory of $\mathcal{T}_{\mathcal{R}}$ defines satisfiability on sequences of database states (paths), whereas execution, defined through the notion of executional entailment, focuses on database sequences. One can say that a formula can be executed on a database sequence, or *equivalently* that the formula is true on the path that corresponds to that sequence. A *split* of a database sequence is defined analogously to a split of a path.

5.1.2 Representing Relations

Evaluation methods for classical Horn logic assign each predicate a relation, which contains all the ground atoms for this predicate that are true. A tuple, $(x_1, \ldots, x_n)$ in the relation for an n-place predicate, p, means that $p(x_1, \ldots, x_n)$ is true. Path-structures assign to each predicate, p, one such relation *for each path*, π, which contains all the atoms for p that are true on π. One way to state that a ground atom, $p(x_1, \ldots, x_n)$, is true on a path, π, is to put a tuple $(x_1, \ldots, x_n, \pi)$ in a relation for p. Since π is a sequence of databases, we can assign a unique identifier to each database, so that we can refer to this database, and store a path as a tuple of such identifiers. We would like each relation to have a fixed number of attributes. This implies that we cannot store paths of different length in the same relation. An updating transaction may be true, however, on paths of arbitrary length. This means that more than one relation might be necessary to store the facts for a predicate.

The relations we use to store facts for predicates are as follows. Relational schemes include a sequence of attributes that accept database identifiers as values. A *database identifier* is a pointer to a database, used for reference to that database. Each relation has a fixed scheme, i.e., it has a fixed number of attributes. Since an updating transaction may cause a database transformation of arbitrary length, each predicate is associated with a *set* of relations. Let p be an n-place predicate. Each relation for p has *at least* $n+1$ columns. Let P^m be a relation for p with $n+m$ columns. P^m consists of the logical consequences for p on paths of length m. We call P^m the *m-th relation for p*. The i-th column, $1 \leq i \leq n$, corresponds to the i-th attribute of p. The other columns store a sequence of database identifiers. The tuple

$$(x_1, \ldots, x_n, \mathbf{D}_1, \ldots, \mathbf{D}_m)$$

in P^m means that $p(x_1, \ldots, x_n)$ is a logical consequence of the program on the path $\mathbf{D}_1, \ldots, \mathbf{D}_m$. Since P^i does not have names for its scheme attributes, we refer to its columns by numbers. We also refer to the $(n + j)$-th column, $1 \leq j \leq i$ as the j-th *db-column*. If $(x_1, \ldots, x_{n+m})$ is a tuple, μ, in P^m, we call x_i the i-th *component* of μ.

Predicates defined only in databases mentioned in the transition base are divided into extensional (EDB) and intensional (IDB), as usual. Relations for EDB predicates are *stored*. Predicates defined in the transaction base are all intensional predicates. Relations for intensional predicates are not stored, and are referred to as *derived*. Temporarily, we make two simplifying assumptions. First, we assume that all the databases mentioned in the transition base are *known*. This means that we know exactly what the stored data and rules are, for each database in the transition base. We refer to these databases through database identifiers. Second, we assume that the relations for elementary updates are *extensional*. This means that all the facts derived directly from elementary transitions are initially available. The goal of a bottom-up evaluation method is to construct the derived relations.

Example 5.1 Consider the following transaction definition,

$(r1)\qquad fae(qa) \leftarrow emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa).$

$(r2)\qquad fae(qa) \leftarrow \neg emp(X, qa).$

where *emp* is an EDB predicate, and *del*:*emp* is a predicate for an elementary update. The atom $emp(name, dept)$ means that *name* is an employee in the *dept* department. The atom $del{:}emp(name, dept)$ removes employee *name* from the *dept* department. Consider the following transition base,

$\langle \mathbf{D}_0, \mathbf{D}_{10} \rangle del{:}emp(john, qa)$

$\langle \mathbf{D}_0, \mathbf{D}_{11} \rangle del{:}emp(jack, qa)$

$\langle \mathbf{D}_{10}, \mathbf{D}_2 \rangle del{:}emp(jack, qa)$

$\langle \mathbf{D}_{11}, \mathbf{D}_2 \rangle del{:}emp(john, qa)$

...

where the databases $\mathbf{D}_0$, $\mathbf{D}_{10}$, $\mathbf{D}_{11}$ and $\mathbf{D}_2$ are as follows.

$\mathbf{D}_0$	$\mathbf{D}_{10}$	$\mathbf{D}_{11}$	$\mathbf{D}_2$
$emp(mike, rd)$	$emp(mike, rd)$	$emp(mike, rd)$	$emp(mike, rd)$
$emp(jack, qa)$	$emp(jack, qa)$	$emp(john, qa)$	
$emp(john, qa)$			

Since emp and $del{:}emp$ are $true$ only on paths of length one and two, respectively, each of these predicates is associated with *one* relation. Let *EMP* and *DEL:EMP* be the respective relations. Following the approach described above, a tuple $(name, dept, \mathbf{D})$ in *EMP* means that $emp(name, dept)$ is a fact in the database $\mathbf{D}$. A tuple $(name, dept, \mathbf{D}_1, \mathbf{D}_2)$ in *DEL:EMP* means that $del{:}emp$ removes successfully the tuple $emp(name, dept)$ from the database $\mathbf{D}_1$, transforming $\mathbf{D}_1$ into $\mathbf{D}_2$. Specifically, the relations *EMP* and *DEL:EMP* are

EMP

$jack$	qa	$\mathbf{D}_0$
$john$	qa	$\mathbf{D}_0$
$mike$	rd	$\mathbf{D}_0$
$jack$	qa	$\mathbf{D}_{10}$
$mike$	rd	$\mathbf{D}_{10}$
$john$	qa	$\mathbf{D}_{11}$
$mike$	rd	$\mathbf{D}_{11}$
$mike$	rd	$\mathbf{D}_2$
...		

$DEL : EMP$

$john$	qa	$\mathbf{D}_0$	$\mathbf{D}_{10}$
$jack$	qa	$\mathbf{D}_0$	$\mathbf{D}_{11}$
$jack$	qa	$\mathbf{D}_{10}$	$\mathbf{D}_2$
$john$	qa	$\mathbf{D}_{11}$	$\mathbf{D}_2$
...			

Since the transition base mentions an infinite number of databases, these relations have infinitely many entries. Predicate fae may be true on paths of any length. Thus, fae is associated with a *set* of relations. Since fae is a 1-place predicate, the n-th relation for fae has $n+1$ columns. Each of these relations is initially (i.e., before execution) empty, since fae is an intensional predicate. □

5.1.3 The Set of Relations Defined by a Rule Body

Consider the rule $\alpha \leftarrow \beta_1 \otimes \ldots \otimes \beta_n$. In proofs using this rule, we must determine the set of values we may substitute for the variables of the rule that make the body true. Let $X_1, \ldots, X_k$ be all the variables appearing in the above rule. Then, we must determine the pairs (θ, π), such that

$$\beta_1\theta \text{ is true on a path, } \pi_1,$$

$$\vdots$$

$$\beta_n\theta \text{ is true on a path, } \pi_n, \text{ and}$$

$$\pi = \pi_1 \circ \ldots \circ \pi_n,$$

where θ is a substitution for $X_1, \ldots, X_k$, and π is a path. If these conditions hold for some substitution, θ, and path, π, then we say that $\beta_1 \otimes \ldots \otimes \beta_n$ *is made true by substitution θ on the path π*. Let b_i be the predicate symbol of subgoal β_i, $X_{i_1}, \ldots, X_{i_s}$ the variables among the X_i's appearing in β_i, and x_j the value substituted by θ for the variable X_{i_j}. Then we say that β_i *is made true by substitution θ on the path* $\mathbf{D}_0, \ldots, \mathbf{D}_m$, if $(x_1, \ldots, x_s, \mathbf{D}_0, \ldots, \mathbf{D}_m)$ is in the m-th relation, B^m, for predicate b.

Example 5.2 Consider the rule

$$(r1) \qquad fae(qa) \leftarrow emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa).$$

in Example 5.1. Its body is made true on a path, π, by a substitution, $X = name$, if:

1. $emp(name, qa)$ is true on π_1.

2. $del{:}emp(name, qa)$ is true on π_2.

3. $fae(qa)$ is true on π_3.

4. $\pi = \pi_1 \circ \pi_2 \circ \pi_3$.

For example, *assume* that the first relation, FAE^1, for fae contains the tuple $(qa, \mathbf{D}_2)$, i.e., that $fae(qa)$ is true on $\mathbf{D}_2$. By looking at the relations *EMP* and *DEL:EMP*, we have that

1. $emp(jack, qa)$ is true on $\mathbf{D}_{10}$.

2. $del{:}emp(jack, qa)$ is true on $\mathbf{D}_{10}, \mathbf{D}_2$.

3. $fae(qa)$ is true on $\mathbf{D}_2$.

4. $\mathbf{D}_{10}, \mathbf{D}_2 = \mathbf{D}_{10} \circ \mathbf{D}_{10}, \mathbf{D}_2 \circ \mathbf{D}_2$.

Therefore, the body of $r1$ is made true by the substitution $X = jack$ on the path $\mathbf{D}_{10}, \mathbf{D}_2$. Similarly, the body of $r1$ is made true by the substitution $X = john$ on the path $\mathbf{D}_{11}, \mathbf{D}_2$. $\Box$

The (substitution, path)-pairs that make a rule, r, true construct a set of relations for that rule, called *the set of relations for rule* r. We define the *m-th relation, R^m, for a rule, r,* to have the scheme $X_1, \ldots, X_k, 1, \ldots, m$, where the X_i's are all the variables of the body of r, in some selected order, and the i's, $i = 1, \ldots, m$, are database identifier attributes. Let r have n subgoals. Then, the tuple $(x_1, \ldots, x_k, \mathbf{D}_0, \ldots, \mathbf{D}_m)$ is in the relation R^m, if and only if there are paths $\pi_1, \ldots, \pi_n$, such that $\mathbf{D}_0, \ldots, \mathbf{D}_m = \pi_1 \circ \ldots \circ \pi_n$, and when we substitute x_i for X_i, $1 \le i \le k$, the j-th subgoal is made true on the path π_j, $1 \le j \le n$.

Computing the Set of Relations for a Rule Body

Consider the rule $\alpha \leftarrow \beta_1 \otimes \ldots \otimes \beta_n$. A substitution, θ, that makes a subgoal, β_i, true, selects a set of tuples from the set of relations for the predicate of β_i. Thus, each subgoal in a rule body is associated with a set of relations. The sets of relations for each β_i, $i = 1, \ldots, n$, are used to construct the set of relations for the rule body. This is achieved through *relational algebra operations* [Ull89a]. The following example is an informal explanation of how the construction of these relations takes place.

Example 5.3 Consider again the rule

$$(r1) \qquad fae(qa) \leftarrow emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa).$$

in Example 5.1. Subgoal $emp(X, qa)$ is associated with *one* relation, namely the set of tuples of the form $(name, \mathbf{D})$, such that $(name, qa, \mathbf{D})$ is in the *EMP* relation. There are two operations involved in the construction of this relation. First, a selection operation, σ, is applied on *EMP*, identifying the tuples $(name, qa, \mathbf{D})$. Second, a projection, π, is applied on the first and third components of the selected tuples, yielding tuples of the form $(name, \mathbf{D})$. Let EMP_{qa} be this relation. Then,

$$EMP_{qa} = \pi_{\$1\$3}(\sigma_{\$2=qa}(EMP)),$$

where $\$i$ represents the i-th component of a tuple. The tuple $(name, \mathbf{D})$ in EMP_{qa} means that $emp(X, qa)$ is made true by the substitution $X = name$ on the path $\mathbf{D}$. Similarly, let $DEL{:}EMP_{qa}$ be the relation for goal $del{:}emp(X, qa)$. Then,

$$DEL : EMP_{qa} = \pi_{\$1\$3\$4}(\sigma_{\$2=qa}(DEL : EMP)).$$

The tuple $(name, \mathbf{D}_1, \mathbf{D}_2)$ in $DEL{:}EMP_{qa}$ means that $del{:}emp(X, qa)$ is made true by the substitution $X = name$ on the path $\mathbf{D}_1, \mathbf{D}_2$. EMP_{qa} and $DEL{:}EMP_{qa}$ are as follows.

EMP_{qa}		$DEL : EMP_{qa}$		
$jack$	$\mathbf{D}_0$	$john$	$\mathbf{D}_0$	$\mathbf{D}_{10}$
$john$	$\mathbf{D}_0$	$jack$	$\mathbf{D}_0$	$\mathbf{D}_{11}$
$jack$	$\mathbf{D}_{10}$	$jack$	$\mathbf{D}_{10}$	$\mathbf{D}_2$
$john$	$\mathbf{D}_{11}$	$john$	$\mathbf{D}_{11}$	$\mathbf{D}_2$
...		...		

Goal $fae(qa)$ is associated with a set of relations. Let FAE^i_{qa} be the i-th relation for $fae(qa)$. Then

$$FAE^i_{qa} = \pi_{\$2 \ldots \$i}(\sigma_{\$1=qa}(FAE^i)).$$

This relation is initially empty, since FAE^i is initially empty. Let $RULE^i_1$ denote the i-th relation for rule $r1$. $RULE^i_1$ has the scheme $X, 1, \ldots, i$. In the following we describe informally how this relation is constructed.

As shown in Example 5.2, *assuming* that that FAE^1_{qa} contains the tuple $(qa, \mathbf{D}_2)$, the body of $r1$ is made true by the substitution $X = jack$ on the path $\mathbf{D}_{10}, \mathbf{D}_2$, and the substitution $X = john$ on the path $\mathbf{D}_{11}, \mathbf{D}_2$. Thus, the tuples $(jack, \mathbf{D}_{10}, \mathbf{D}_2)$ and $(john, \mathbf{D}_{11}, \mathbf{D}_2)$ are in $RULE^2_1$. These tuples can be derived from the relations EMP_{qa}, $DEL{:}EMP_{qa}$ and FAE^1_{qa} through *natural joins*. This is accomplished by *naming* the attributes of the schemes of these relations. In particular, we name the first columns of EMP_{qa} and $DEL{:}EMP_{qa}$ after the variable X, and assign *numbers* as names to the db-columns. We assign EMP_{qa} the scheme $X\ 1$, $DEL{:}EMP_{qa}$ the scheme $X\ 1\ 2$, and FAE^i_{qa} the scheme $2\ 3 \ldots i+1$. Note that in this numbering, the name of the db-column of EMP_{qa} is the same as that of the first db-column of $DEL{:}EMP_{qa}$, and the name of the second db-column of $DEL{:}EMP_{qa}$ is the same as the name of the first db-column of FAE^i_{qa}. Our objective is to simulate path composition. This is done through *natural joins*. In particular, $RULE^i_1$ is the natural join (denoted as $\bowtie$) of EMP_{qa}, $DEL{:}EMP_{qa}$, and FAE^i_{qa}:

$$
\begin{array}{cc} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \\ \multicolumn{2}{c}{\ldots} \end{array}
\bowtie
\begin{array}{ccc} \multicolumn{3}{c}{DEL: EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{3}{c}{\ldots} \end{array}
\bowtie
\begin{array}{c} FAE^1_{qa} \\ 2 \\ \mathbf{D}_2 \\ \ldots \end{array}
=
\begin{array}{ccc} \multicolumn{3}{c}{RULE^2_1} \\ X & 1 & 2 \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{3}{c}{\ldots} \end{array}
$$

□

In [Ull89a] an algorithm is given for computing the relation for a safe Datalog[1] rule body, using relational algebra operations. The algorithm handles both ordinary and built-in goals. Its input is a rule, r, and a set of relations, $R_1, \ldots, R_n$. R_i is a relation for the predicate of the i-th subgoal. Its output is an expression of relational algebra, called EVAL-RULE$(r, R_1, \ldots, R_n)$, that computes the relation for the body of r. The algorithm works in four steps: For each ordinary subgoal, p_i, in r's body, the first step of the algorithm selects from R_i those tuples that make p_i true, and applies a projection on the variables of p_i. The result, Q_i, is a relational algebra expression. The second step handles variables not found among ordinary subgoals. Since r is safe, each such variable, X, is equated to another variable that takes on a finite number of values. This step computes for X a unary relation, D_X, containing all the values that X could have in a substitution that satisfies all the subgoals of r. The third step is the natural join of the relations computed in the first two steps. Q_i is regarded as a relation whose attributes are the variables in p_i, and D_X is a relation with attribute X. The last step computes a selection, $\sigma_F(E)$, where F is the conjunction of $X\theta Y$, for each built-in subgoal $X\theta Y$ in r's body, and E is the expression constructed in the previous step. $\sigma_F(E)$ is the output expression, i.e., EVAL-RULE$(r, R_1, \ldots, R_n) = \sigma_F(E)$.

[1]Datalog is a subset of classical Horn logic with no function symbols.

The approach described informally in Example 5.3 is similar to this algorithm. It differs in that we deal with sets of relations, and use extra arguments for ordinary subgoals to represent paths. To compute the set of a relations for a rule body, we adopt this algorithm *as it is*. However, the relations passed to the arguments of EVAL-RULE must *have their db-columns named.*

The computation of the set of a relations for a rule body is as follows. Let r be a rule with n ordinary subgoals, $\beta_1, \ldots, \beta_n$, and p_i be the predicate of β_i. Let the relations for p_i be $R_i^1, \ldots, R_i^{j_i}$. First, *choose an order* for the predicates of the subgoals of r. For the time being, we are not concerned how this order is chosen, and whether this action is sound. We cover this issue in Section 5.2.4. Let the chosen order be $p_1, \ldots, p_n$. Then, the naming takes place as shown in Figure 5.1. This figure also shows the algorithm, EVAL-RULE$^{\mathcal{T_R}}$, for computing the set of relations for a rule body. The arguments to EVAL-RULE$^{\mathcal{T_R}}$ are a rule, r, and the sets of relations for the predicates of the subgoals of r, *in some selected order.*

5.1.4 Computing the Meaning of Rules

Bottom-up evaluation based on the framework we set in the previous sections is the same as bottom-up evaluation for Datalog programs. The only difference is that we compute a set of relations for each rule body. Thus, both the Naive and Semi-naive evaluation methods [Ull89a] are applicable.

In brief, we start with a set of stored relations, $R_1, \ldots, R_n$, and a set of sets of derived relations, $P_1, \ldots, P_m$. The latter are initially empty. The goal of bottom-up evaluation is to compute the derived relations. Computation involves several iterations. For each rule, r, each iteration computes the set of relations for the body of r (using EVAL-RULE$^{\mathcal{T_R}}$), and *projects* each relation in that set on the variables of the head *and the db-columns.* Let p_i be a derived predicate appearing in the heads of rules $r_1, \ldots, r_k$, and $Q_1, \ldots, Q_k$ be the sets of relations for p_i returned by EVAL-RULE$^{\mathcal{T_R}}$ when applied to $r_1, \ldots, r_k$. Then, the i-th relation for p_i is the *union* of the relations in $Q_1, \ldots, Q_k$ with i db-columns.

procedure name($R_1, \ldots, R_n$)
begin
Let $max(R_i)$ denote the number of db-columns of R_i.
Name the db-columns of R_i as follows.

1. Assign to the db-columns of R_1 the scheme $1\ 2 \ldots max(R_1)$.
2. Let m be the name of the last column of R_{i-1}.
   Assign to the db-columns of R_i, $i = 2, \ldots, n$, the scheme $m\ m+1 \ldots m+max(R_i)$,

end

procedure EVAL-RULE$^{\mathcal{T}_\mathcal{R}}(r, S_1, \ldots, S_n)$
begin
Let the relations in the set S_i be $R_i^1, \ldots, R_i^{j_i}$, $i = 1, \ldots, n$.
for $i_1 = 1, \ldots, j_1$
  for $i_2 = 1, \ldots, j_2$
    ...
    for $i_n = 1, \ldots, j_n$
    begin
      name($R_1^{i_1}, \ldots, R_n^{i_n}$).
      EVAL-RULE($r, R_1^{i_1}, \ldots, R_n^{i_n}$).
    end
end

Figure 5.1: The algorithm for computing the set of relations for a rule body.

(Recall that we consider rectified rules. Thus, these relations have the same scheme.)

The following example is an application of the Semi-naive evaluation method. Semi-naive evaluation works as described above, except that each iteration avoids to recompute facts computed in previous steps.

Example 5.4 Consider again the rules of Example 5.1

$$(r1) \qquad fae(qa) \leftarrow emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa).$$

$$(r2) \qquad fae(qa) \leftarrow \neg emp(X, qa).$$

with initial database $\mathbf{D}_0$. According to our approach, the relations for the subgoals

$emp(X)$ and $del{:}emp(X)$ are

EMP_{qa}		$DEL : EMP_{qa}$		
$jack$	$\mathbf{D}_0$	$john$	$\mathbf{D}_0$	$\mathbf{D}_{10}$
$john$	$\mathbf{D}_0$	$jack$	$\mathbf{D}_0$	$\mathbf{D}_{11}$
$jack$	$\mathbf{D}_{10}$	$jack$	$\mathbf{D}_{10}$	$\mathbf{D}_2$
$john$	$\mathbf{D}_{11}$	$john$	$\mathbf{D}_{11}$	$\mathbf{D}_2$
...		...		

and the set of relations for fae is initially empty. Figure 5.2 shows how the facts for fae are derived, by using the Semi-naive Evaluation method. The first iteration yields the tuple $(\mathbf{D}_2)$ for FAE^1_{qa}, by using rule $r2$. The applied operation is a *set difference*.[2] The second iteration uses the facts derived in the first iteration to generate the new tuples $(\mathbf{D}_{10}, \mathbf{D}_2)$ and $(\mathbf{D}_{11}, \mathbf{D}_2)$ for FAE^2_{qa}. These new tuples are used in the third iteration to derive the tuples $(\mathbf{D}_0, \mathbf{D}_{10}, \mathbf{D}_2)$ and $(\mathbf{D}_0, \mathbf{D}_{11}, \mathbf{D}_2)$. These tuples specify two possible executions on $\mathbf{D}_0$ for the transaction $? - fae(qa)$. $\Box$

5.2 Towards a Realistic Approach

There are three reasons why the above approach is not practical. First, it materializes the entire least fixpoint. Therefore, no finite number of iterations suffices for the method to terminate, and the demand for space is unlimited. Second, because there are infinitely many databases, each relation has an *infinite* number of entries. Finally, both the representation of the relations for updates as stored relations and the assumption that all possible databases are known are *unrealistic*.

In this section we establish the basis for a realistic bottom-up evaluation method. We discuss how the transition base is likely to be implemented, we describe a possible scenario for the implementation of elementary updates, and specify our framework for update execution.

[2]The reader may take the result of the first iteration for granted. Negation is not a concern of our work. We merely use it to make this example realistic.

$$\textbf{1.}\quad \begin{array}{ll} \multicolumn{2}{c}{DOM_{qa}} \\ X & 1 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \\ john & \mathbf{D}_2 \\ jack & \mathbf{D}_2 \\ \multicolumn{2}{c}{\ldots} \end{array} \; - \; \begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \\ \multicolumn{2}{c}{\ldots} \end{array} \quad \stackrel{\pi_{\$2}}{=} \quad \begin{array}{c} FAE^1_{qa} \\ 1 \\ \mathbf{D}_2 \\ \ldots \end{array}$$

$$\textbf{2.}\quad \begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \\ \multicolumn{2}{c}{\ldots} \end{array} \; \bowtie \; \begin{array}{lll} \multicolumn{3}{c}{DEL : EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{3}{c}{\ldots} \end{array} \; \bowtie \; \begin{array}{c} FAE^1_{qa} \\ 2 \\ \mathbf{D}_2 \\ \ldots \end{array} \quad \stackrel{\pi_{\$2\$3}}{=} \quad \begin{array}{ll} \multicolumn{2}{c}{FAE^2_{qa}} \\ 1 & 2 \\ \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{2}{c}{\ldots} \end{array}$$

$$\textbf{3.}\quad \begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \\ \multicolumn{2}{c}{\ldots} \end{array} \; \bowtie \; \begin{array}{lll} \multicolumn{3}{c}{DEL : EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{3}{c}{\ldots} \end{array} \; \bowtie \; \begin{array}{ll} \multicolumn{2}{c}{FAE^2_{qa}} \\ 2 & 3 \\ \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{2}{c}{\ldots} \end{array} \quad \stackrel{\pi_{\$2\$3\$4}}{=} \quad \begin{array}{lll} \multicolumn{3}{c}{FAE^3_{qa}} \\ 1 & 2 & 3 \\ \mathbf{D}_0 & \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_0 & \mathbf{D}_{11} & \mathbf{D}_2 \\ \multicolumn{3}{c}{\ldots} \end{array}$$

Figure 5.2: Figure for Example 5.4. Semi-naive Evaluation.

5.2.1 The Transition Base

The transition base is a theoretical construct, that allows for a clean integration of updates and logic. This is accomplished through the following two, nice properties. First, it specifies all valid databases for a given program. Second, it specifies fully the behaviour of elementary updates. In practice, an implementation of transition base will not have these properties. In the sequel, we discuss the role of the transition base in a real setting, focusing on the properties of elementary updates, and define some terms used in the rest of this thesis.

Databases

Recall that transaction execution is defined through the notion of executional entailment. Given a $\mathcal{T_R}$ program, $P = (\mathcal{B}, \mathbf{P}, \mathbf{D}_0)$, a transaction, $\leftarrow \alpha$, can be executed successfully if

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n \models \exists\alpha,$$

for some databases $\mathbf{D}_1, \ldots, \mathbf{D}_n$. Before execution starts, only the initial database, $\mathbf{D}_0$, is given. $\mathbf{D}_0$ is said to be *known*. Transaction execution transforms $\mathbf{D}_0$ through the intermediate databases $\mathbf{D}_1, \ldots, \mathbf{D}_{n-1}$ to the final database $\mathbf{D}_n$. Initially, these databases are said to be *unknown*. We assume that during transaction execution there is a set of *known* databases, initially containing only $\mathbf{D}_0$. We refer to this set as the *pool* of databases. Databases are added to this pool, by performing updates on the databases that are already in the pool. Thus, if database $\mathbf{D}_i$ is known, $\mathbf{D}_{i+1}$ becomes known after the execution of an elementary update on $\mathbf{D}_i$ is complete.

Elementary Updates

An interpreter for Horn–$\mathcal{T_R}$ could use both built-in and user defined updates. The latter are likely to be defined as *procedures* in a higher-level programming language. Thus, we assume that each elementary update is an independent module with an Input/Output interface, attached to the evaluation procedure. These modules could even be vendor supplied, satisfying user specifications.

Each elementary update receives some input, returns some output results, and has a side effect. The input to an elementary update is a set of values for some of its arguments, and a database. We refer to this database as the *db-target* of the update. The update executes on its db-target, and returns a value for each of its free arguments as output. Its side effect is a transformation of its db-target. For the rest of this thesis, we regard the side effect of a transaction as the union of the sets of tuples inserted and deleted by that transaction. A tuple may be a unit or a definite clause.

Definition 5.1 (*Side effect of a transaction*) The side effect of a transaction, α, on a database, $\mathbf{D}$, is defined to be the set

$$SD^\alpha = \{+t : \text{tuple } t \text{ is added to } \mathbf{D} \text{ by } \alpha\} \cup \{-t : \text{tuple } t \text{ is removed from } \mathbf{D} \text{ by } \alpha\}.$$

□

Thus, each transaction behaves as an elementary update. Queries, which have no side effects, are considered to have an null side effect. We refer to the side effect and the returned values of a transaction as the *result* of that transaction.

Elementary updates may be non-deterministic. A *non-deterministic* update has a ground instance which maps some database to more than one databases. *Deterministic* updates are one-to-one mappings. Both deterministic and non-deterministic updates may give rise to non-deterministic execution. The following is an example of non-deterministic execution of a deterministic update.

Example 5.5 (*Non-deterministic execution*) Consider the following transaction definition.

$$(r1) \qquad hire(Dept) \leftarrow qualified(X) \otimes ins{:}hired(X, Dept).$$

Subgoal $qualified(X)$ returns a set of values for variable X, that specifies a group of qualified individuals. A ground instance, $ins{:}hired(x, dept)$, of the subgoal $ins{:}hired(X, Dept)$ adds a tuple to the database, of the form $hired(x, dept)$, meaning that individual x has been hired in the department $dept$.

Consider now the transaction $? - hire(is)$. Let S be the set of values returned by the subgoal $qualified(X)$. The update $ins{:}hired(x, is)$ performs a different database transformation, for each value x in S. According to the semantics of the logic, only one execution path is chosen. Thus, only one value is chosen from this set. Hence, the result of $? - hire(is)$ is the hiring of *some* qualified individual in the is department.

Consider now the transaction $? - hire(Dept)$. Then, the second argument of the update is free. The result of this transaction is the hiring of *some* qualified individual in *some* department. □

Updates change only *stored* data. We do *not* allow updates of derived relations. A view update would make sense if it targets at the data that define the view. We do not, however, take into account such updates. Thus, whenever a database transformation occurs, we need only specify the changes of stored data. Changes are then propagated to views through the database rules. An elementary update may be defined, however, on *derived* data. This means that an update may change the stored data of its target database with respect to derived relations in this database. Recall that Horn–$\mathcal{T}_\mathcal{R}$ programs allow a transaction to be defined through both transaction base and database rules. Elementary updates on derived data do *not* take into account the rules of the transaction base.

Example 5.6 (*Updates on derived data/Bulk updates*) Updates of the form $[r := q]$ are called *bulk* updates [BK93]. The update $[r := q]$ copies the contents of the relation for q into the relation for r. Previous contents of the relation for r are lost.

Consider the following rule definitions in the database

$$(r1) \qquad sg(X, X).$$

$$(r2) \qquad sg(X, Y) \leftarrow par(X, X1) \ \& \ sg(X1, Y1) \ \& \ par(Y, Y1).$$

and the rule

$$(r3) \qquad sg\_t(Y) \leftarrow [a\_sg := sg(a, Y)].$$

in the transaction base. Rules $r1$ and $r2$ compute cousins at the same generation. The transaction $? - sg\_t(Y)$ copies to the relation for a_sg all tuples (a, y), such that y is at the same generation as a. Note that, in order that this update is executed correctly, the relation for sg must be *fully* evaluated. $\square$

5.2.2 Executing Elementary Updates

We can now drop the assumption made in Section 5.1.2 about the relations for elementary updates being stored relations. Since elementary updates are implemented through procedures in some higher-level language, the relation for an elementary update in a rule body is *computed* through a *procedure call*, with arguments the arguments of the elementary update. The procedure must also be given the db-target of the update. For pragmatic reasons, we are taking the following approach in the execution of elementary updates.

Assumptions and Requirements on the Execution of Elementary Updates

We handle only *deterministic* elementary updates. A ground instance of a deterministic update has the property that it maps its db-target to *one* database. On the contrary, a ground instance of a non-deterministic update may choose among a number of possible mappings. For a given db-target, some of the database transformations of that update might lead to failure. If one transformation leads to failure, another must be tried, until one that leads to success is found, or all alternatives are exhausted. The question is how to implement this sort of if-then-else execution. One solution would be to have the update execute in all possible ways. This would be clearly inefficient. An alternative would be to design the interface of non-deterministic updates such that we can let the update know that a certain execution should not be considered again (because it has been already performed). The problem is non-trivial. We thus handle only deterministic updates, and leave consideration of non-deterministic updates to future research.

We require that *all* arguments of a procedure call for an elementary update receive input values. This requirement is imposed for the following two reasons. First, a deterministic update with free arguments behaves *non-deterministically*. In Example 5.5, the transaction $? - hire(Dept)$ invokes the elementary update $ins{:}hired(X, Dept)$ with its second argument free. This means that the returned value for the variable *Dept* is chosen non-deterministically. Thus, we face the same problem discussed above. If one returned value leads to failure, we must be able to get another value, i.e., execute the update in a different way. Second, in theory updates with free arguments are executed by choosing an elementary transition in the transition base whose head unifies with the update. In practice, there is no such thing as "elementary transition". The behaviour of an elementary update is specified in the finite code of its procedure. For software engineering purposes, restrictions might be imposed on the arguments of an update, specifying that some arguments are not allowed to be free. For these two reasons, we consider a call to an elementary update valid, only if it binds all its arguments.

Executing Updates with all Arguments Bound

Now we face the problem of computing the relations for elementary updates through procedure calls. The problem is hard for the following reasons. First, we have set a requirement for executing elementary updates with all arguments bound. This means that prior to the execution of an elementary update a set of values for its arguments has to be computed. A first implication is that for each elementary update there is some source that provides its input, i.e., the values for its arguments. Another implication is that the computation of the set of relations for a rule body is broken into pieces. This is because the relation for an elementary update, u, must wait for the values of its arguments to be computed. Second, u must be provided with a set of db-targets. The question that arises is how the databases in this set are chosen from the pool of known databases.

The following example discusses these problems, and shows how bottom-up evaluation must be modified to conform with the guidelines of Sections 5.2.1 and 5.2.2.

Example 5.7 Consider again the program of Example 5.4 again.

$$(r1) \qquad fae(qa) \leftarrow emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa).$$

$$(r2) \qquad fae(qa) \leftarrow \neg emp(X, qa).$$

Since at first only the initial database, $\mathbf{D}_0$, is given, the set of relations for fae is empty, while EMP_{qa} is

EMP_{qa}	
$jack$	$\mathbf{D}_0$
$john$	$\mathbf{D}_0$

Clearly, the first argument of $del{:}emp(X, qa)$ must receive bindings from the relation for $emp(X, qa)$. EMP_{qa} is thus the first relation to be computed. Figure 5.3 shows how we expect semi-naive bottom-up evaluation to take place. Both tuples in EMP_{qa} are passed as input to the update. Thus, in the first iteration, the update $del{:}emp$ is executed twice, with input $(john, \mathbf{D}_0)$ and $(jack, \mathbf{D}_0)$, respectively. This means that the db-target for both executions of $del{:}emp(X, qa)$ is database $\mathbf{D}_0$. The execution of $del{:}emp(X)$ is not shown explicitly in the figure. The assumption is that it is triggered before the join operation takes place. The join is performed after $del{:}emp(X)$ completes execution for all inputs. *DEL:EMP* is regarded as a *computed* relation, since it is produced through update execution. Although no facts for the set of relations for fae are generated in the first two iterations, the evaluation continues. This is because *new databases are derived*. Thus, the question is whether we should change the termination criterion to take into account the generation of new databases arises. Section 5.3 provides an answer to this question. □

5.2.3 Selecting Relevant Rules

Given a program and a transaction to be executed, only a portion of the program is *relevant* to the transaction, i.e., it contributes to the result of the transaction. In classical Horn logic, bottom-up evaluation of a given query is done with respect to the relevant portion of the program. This portion is determined by inspecting the *dependency graph* of the program [Ull89a]. The dependency graph of a program, P, is a graph whose nodes are the ordinary predicates. There is an arc from a predicate, p, to a predicate, q, if there is a rule with a subgoal whose predicate is p and with a head whose predicate is q. If the predicate of a given query is q, then the paths in the dependency graph which end at q specify all the predicates that are taken to be relevant to the given query. A rule is taken to be relevant, if the predicate of its head is relevant.

There are two issues related to the selection of relevant rules in Horn–$\mathcal{T}_\mathcal{R}$, that arise from database changes. The first regards database rule changes. The other has to do with update definitions on derived data.

Database Rule Changes

Transaction logic allows updates of database rules. Thus, the set of rules in the initial database may change during transaction execution. During top-down evaluation, it is possible that a rule which seems to be relevant to a transaction is deleted, before it is actually used in evaluation. In effect, that rule is not relevant to that transaction. Similarly, it is possible that at some point during execution, a rule relevant to a transaction is inserted. Apparently, both of these situations cannot be remedied by inspecting the dependency graph at compile-time.

We take the simplified approach of dealing with transactions that do *not* cause rule changes. In other words, we are making the following assumption.

- The set of relevant rules remains *unchanged* throughout the lifetime of a transaction.

The issue of rule changes remains as a future consideration.

Update Definitions on Derived Data Revisited

As discussed in Section 5.2.1, an elementary update may change the stored data with respect to derived data. For an update to be executed correctly, we must ensure that the derived data it uses have been fully evaluated, when that update is executed. This means that the rules for derived data, which are used by elementary updates that are relevant to a given query, must be included in the set of relevant rules.

For this reason, we extend the definition of dependency graph for Horn–$\mathcal{T}_\mathcal{R}$ as follows.

Definition 5.2 (*Dependency graph*) The dependency graph of a Horn–$\mathcal{T_R}$ program, P, is a graph whose nodes are the ordinary predicates. There is an arc from a predicate, p, to a predicate, q, if there is a rule with a subgoal whose predicate is p and with a head whose predicate is q, or if q is an elementary update predicate which reads data of the relation for p. $\square$

5.2.4 The Free Execution Order Property

The ordering of the subgoals in rules bodies is an important issue in transaction execution. In Horn logic, any ordering yields the same result, since all subgoals are queries, i.e., they have no side effects. Bottom-up evaluation orders the subgoals such that joins are as efficient as possible. Top-down evaluation uses heuristics to choose for evaluation a subgoal which is likely to have the lower cost. In Horn–$\mathcal{T_R}$, however, ordering flexibility is limited due to the presence of updates. Updates create dependencies between transactions. We give the definition of *independence* for transactions analogously to the definition of independence of queries and updates in [Elk90].

Definition 5.3 (*Independence*) We call a transaction, t_1, independent of another transaction, t_2, if t_2 cannot change the result of t_1. We call transactions $t_1, \ldots, t_n$ independent, if each one is independent of the others. $\square$

Example 5.8 (*Independence*) Consider the following rules

$$(r1) \qquad ir(X,Y) \leftarrow edge(X,Z) \;\&\; ir(Z,Y).$$

$$(r2) \qquad ir(X,Y) \leftarrow edge(X,Y) \otimes ins{:}arc(X,Y).$$

where $ins{:}arc(x,y)$ inserts the tuple $arc(x,y)$ into the database. Clearly, $ir(Z,Y)$ is an updating goal. It does not, however, affect the relation *EDGE*. Therefore, the goals $edge(X,Z)$ and $ir(Z,Y)$ are independent. This means that we are free to choose any order for $edge(X,Z)$ and $ir(Z,Y)$ in the first rule. $\square$

If transactions $t_1, \ldots, t_n$ are independent, then their execution yields the same final result for any order. Otherwise, t_1 and t_2 must be serialized. Unfortunately, independence of queries and updates is undecidable [Elk90]. Therefore, transaction independence is also undecidable. Deciding independence of subgoals in rule bodies is an important issue for our work, for the following reasons.

Assume that t_1 and t_2 are two *not* independent transactions. The serial conjunction $t_1 \otimes t_2$ specifies an execution order of t_1 and t_2, namely from left to right. Given the goal t_1 & t_2, however, there are two possible executions, corresponding to the serial conjunctions $t_1 \otimes t_2$ and $t_2 \otimes t_1$. It is possible that one leads to failure and the other to success. This means that we must consider both execution orders for t_1 & t_2. Assume now that t_1 and t_2 are independent transactions. Then, it suffices to choose *one* execution order for t_1 & t_2. We would like to remove the task of deciding independence from the evaluation method. Specifically, we would like the evaluation method to be able to choose *any one* execution order for each rule body. The choice of only one order would avoid the complication and drudgery of trying different orders when one leads to failure. In addition, the flexibility in choosing any order allows for optimizations in this selection. This is particularly true for top-down evaluation.

The problem exists due to the presence of the shuffle operator. To accomplish our objective, we impose the following requirement.

- A list, t_1 & ... & t_n, of goals to be executed must have the property that $t_1, \ldots, t_n$ are *independent*.

A rule whose body satisfies this requirement is said to have the *free execution order property*. A program is said to have the free execution order property, if every rule in the program has this property. Every Horn–$\mathcal{T_R}$ program can be rewritten to satisfy this property. For example, given the rule

$(r1) \qquad t \leftarrow t_1 \ \& \ t_2.$

where t_1 and t_2 are not independent (or we cannot decide independency), we create the rules

$$(r2) \qquad t \leftarrow t_1 \otimes t_2.$$

$$(r3) \qquad t \leftarrow t_2 \otimes t_1.$$

and remove $r1$.

5.2.5 Giving Priority to Database Predicates

As mentioned in Section 5.2.1, in order to assure correct execution of updates, all derived relations read by an update must be fully evaluated prior to the execution of that update. To avoid complicated solutions, we choose to give evaluation *priority* to *database relations*. Thus, we split each iteration of the bottom-up evaluation method into two stages. The first stage evaluates completely (i.e., until no more facts are derived) the database rules. When this stage is finished, the rules in the transaction base are evaluated, as usual, and the next iteration starts.

This approach is more efficient, when the semi-naive evaluation method is used. The added effectiveness results from the evaluation of the rules in the transaction base. To see this, consider the rule body $idb \otimes rest$ in the transaction base, where idb is a goal with an IDB predicate, and $rest$ is the rest of the body. The relation of this rule body is evaluated by joining the relation IDB for the goal idb with the relations for the rest subgoals of the body. New tuples added in IDB give rise to new joins. The number of joins is *smaller* if all tuples for IDB are available when the above body is evaluated. In this way, joins become more massive, i.e., they are computed on larger relations. However, the cost of joins in the first case is higher than the cost of computing a small number of massive joins, in the second case [Ull89b].

5.3 Adding Top-Down Functionality

Bottom-up processing exhibits two kinds of data flow. The first is *sideways* binding passing from one subgoal to another, carried out by joins. The other is *bottom-up* binding passing, from a rule body to its head. In addition, top-down processing passes binding information from a rule head to its body (*top-down* binding passing). These three kinds of information flow allows top-down processing to *focus on data relevant to a given query*. Rule rewriting techniques for Horn logic allow bottom-up evaluation to simulate top-down evaluation.

Adding top-down control to bottom-up evaluation of Horn–$\mathcal{T}_{\mathcal{R}}$ programs is crucial for achieving reasonable performance. We show that we can add such control into bottom-up evaluation of Horn–$\mathcal{T}_{\mathcal{R}}$ programs through existing rule rewriting techniques. Before we discuss the necessity for adding top-down control, and show how this can be accomplished through existing rule rewriting methods, we describe the Magic Sets transformation method [BMSU87, Ull89b], which is the most general of these methods.

The Magic Sets Transformation

We will use a classical example to explain the basic principles of the method.

Example 5.9 Consider the following Horn logic program, for computing cousins at the same generation.

$$(r1) \quad sg(X, X).$$

$$(r2) \quad sg(X, Y) \leftarrow par(X, X1) \;\&\; sg(X1, Y1) \;\&\; par(Y, Y1).$$

Consider the query $sg(a, Y)$. A bottom-up evaluation of this program produces the set of all facts of the form (x, y), which is much larger than the set of facts of the form (a, y). On the contrary, a top-down evaluation produces only facts for the predicate sg that are relevant to the given query, i.e., they contribute to the answers of the query. Top-down evaluation is said to *focus on relevant data*. Assume that top-down evaluation processes the goals in the body of rule $r2$ from left to right. Then, a top-down evaluation strategy for $sg(a, Y)$ first passes binding information for variable X (i.e., a) from the head to the leftmost subgoal, $par(X, X1)$. Then, evaluation of $par(a, X1)$ binds $X1$ and passes the new binding information sideways to the subgoal $sg(X1, Y1)$, binding its first argument, X_1. Since sg is recursive, several calls to the subgoal $sg(X1, Y1)$ occur, every time with new bindings. Every call has the first argument bound and the other free. Thus, these calls have the same binding pattern,[3] namely bf.

The *magic set*, for sg^{bf} is the set of bindings for the first argument of the calls to $sg(X1, Y1)$. Let M_SG be the magic set for sg^{bf}, with corresponding predicate m_sg. M_SG has only one column. Let also PAR be the relation for par. Tuple (a) is in M_SG, since a is provided by the top query, $sg(a, Y)$. Looking at rule $r2$ we can see that, if (x) is in M_SG and (x, y) is in PAR, then y binds the first component of $sg(X1, Y1)$. Thus, the tuple (y) is also in M_SG. This reasoning is materialized in the following rules.

$(r3)\quad m\_sg(a).$

$(r4)\quad m\_sg(X1) \leftarrow m\_sg(X)\ \&\ par(X, X1).$

We exploit the magic set for sg^{bf} to rewrite the program, such that the new program is more efficient when evaluated bottom-up. In particular, the new program generates a smaller number of facts of the form $sg(x, y)$. It generates, however, all the facts of the form $sg(a, y)$, i.e., it returns the same answer to the query $? - sg(a, Y)$ as the original program. In effect, the new program generates a smaller amount of irrelevant data. This is achieved by rewriting rules $r1$ and $r2$ to insist that values of the first argument of sg are in its magic set:

$(r1')\quad sg(X, X) \leftarrow m\_sg(X).$

$(r2')\quad sg(X, Y) \leftarrow m\_sg(X)\ \&\ par(X, X1)\ \&\ sg(X1, Y1)\ \&\ par(Y, Y1).$

[3]The binding pattern for a goal, p, with n arguments is a string of b's and f's of length n. If the i-th symbol is b, then the i-th argument of p is bound; otherwise it is free.

The new program to be evaluated bottom-up consists of the rules $r3$, $r4$, $r1'$ and $r2'$. □

The Magic Sets technique described in [Ull89b] transforms a program *with respect to a given query*. The new program evaluates bottom-up the given query more effectively than the original. The Magic Sets transformation requires that the binding pattern for each predicate is unique, with respect to this query (the *unique binding property* [Ull89b]). A program that does not have this property can be rewritten to have it [Ull89b]. The method creates two sets of predicates, namely *magic predicates* and *supplementary predicates*. As discussed in Example 5.9, for each IDB predicate, p, a magic predicate m_p is created, whose arguments are the bound arguments of p (p has the unique binding property). Supplementary predicates are used for efficiency reasons and clarity. For each rule, r_j, with k subgoals, the technique creates the supplementary predicates $sup_{j.i}$, $i = 0, 1, \ldots k-1$. The arguments of $sup_{j.i}$ are all the variables of rule r_j that are either a) bound in the head, or b) appear in one of the first i subgoals of r_j, and also appear in a later subgoal or in the head. Thus, the bindings for the i-th subgoal of a rule, r_j, with an IDB predicate, p, are given by the relation for $sup_{j.i-1}$. The new program, when executed bottom-up, simulates a top-down evaluation of the given query using the original program.

5.3.1 The Necessity for Adding Top-Down Functionality

The ability to focus on relevant data is highly desirable for good performance. Rule rewriting transformations seem to have no problem to be applied to Horn–$\mathcal{T}_{\mathcal{R}}$. However, applying such techniques on updating transactions does not necessarily result in good performance. In fact, such techniques are necessary for satisfying our requirements on the execution of elementary updates. The following examples introduce the problems.

Example 5.10 Termination in Example 5.7 is guaranteed due to the finite number of databases and the acyclic structure of paths, which prevents infinite loops. We cannot claim that computation is efficient, however, since it computes all possible path-answers; but at least it does not generate irrelevant data.

Now, consider the transaction definition

$$hire(marketing) \leftarrow qualified(X) \otimes ins{:}hired(X, marketing).$$

and the query goal $? - hire(marketing)$. Let *QUALIFIED* be the relation for predicate $qualified$, and $|QUALIFIED|$ the size of this relation. Starting from initial database, say $\mathbf{D}_0$, the first iteration of bottom-up evaluation executes $ins{:}hired(x, marketing)$ once for each $x \in QUALIFIED$, yielding $|QUALIFIED|$ path-answers, and generating an equal number of databases. Evaluation should not consider these databases for further execution. If it does, these databases will be *redundant db-targets* for the above two subgoals. This is in contrast with the situation in Example 5.7, where the generation of new databases suggested that computation should continue. If the bottom-up evaluation continues with the new databases, eventually it terminates, after all employees have been hired. All path-answers but those of the first iteration, however, are irrelevant. □

Example 5.11 Consider again the goal $? - hire(marketing)$, but with the following definition for $hire$.

$$hire(Dept) \leftarrow qualified(X) \otimes ins{:}hired(X, Dept).$$

Bottom-up evaluation of this program would not satisfy our requirement for executing elementary updates with bound arguments, because the binding for the second argument of the update comes from the head. In this case, there is a trivial solution: push the constant from the head to the body, rewriting the above rule as follows.

$$hire(marketing) \leftarrow qualified(X) \otimes ins{:}hired(X, marketing).$$

Straightforward solutions are not, however, always possible, especially for recursive rules. This is because most of the bindings are provided at run-time (recall Example 5.9). Thus, *top-down* (i.e., from a rule head to its body) binding passing is often vital for setting bottom-up evaluation feasible. □

These examples have introduced two problems. The Magic Sets technique solves the problem of passing binding information for variables of the head to the body; it does not give an answer, however, to the question of what are the db-targets of each subgoal. In other words, assuming that a transaction, t, is part of a composite transaction that performs several database transformations, how do we know on which database(s) t must be executed? Evaluating t on all derived databases would be naive and costly. Furthermore, it could jeopardize termination, even in non-recursive cases. This problem is analogous to the problem of passing goals binding information for variables. We need to provide each goal with binding information specifying a set of database identifiers to be db-targets for that goal. Thus, the problem reduces to using rule rewriting techniques for adding the necessary control of passing bindings for the db-target of each goal. We call such bindings *database bindings*.

5.3.2 Adding Database Arguments to Ordinary Atoms

In order to use a rule rewriting technique, such as Magic Sets, for passing database bindings, we must be able to refer to the db-target and db-output of each goal *explicitly.* Recall that we are using an internal representation for paths. Paths are part of the relation schemes for goals, but they are not coded into the program. To show them explicitly in the program, we extend each goal with two arguments, one for its db-target and one for its db-output. We refer to these arguments of an atom as its *db-target* and *db-output arguments*. We call variables for these arguments *database variables* (*db-target* and *db-output variables*). They are to take on database identifiers as values. This extension is carried out by the following algorithm.

Algorithm: Extending ordinary atoms in rules with database arguments.
Input: A set of rules that have the free execution order property.
Output: A new set of rules such that each ordinary atom has a db-target and db-output argument.
Method: The following steps are applied to each rule in the given set.

1. Replace each rule body with a serial conjunction. (This is sound, since each rule

has the free execution order property.)

2. To show explicitly the db-target and db-output of each goal, extend all rule heads and all ordinary subgoals in each rule body with one or two arguments. Extend query atoms with one argument, since the db-output of a query is the same as its db-target; extend transactions that possibly perform updates with two arguments.

   For each rule, $r \equiv \alpha \leftarrow \beta_1 \otimes \ldots \otimes \beta_n$,[4] assign database variables to database arguments as follows.

   (a) Choose a *fresh* db-target variable for the head, α, of r.

   (b) The db-target variable of β_1 is the same as the db-target variable for α.

   (c) If β_i, $i = 1, \ldots, n$, has two database arguments, the db-output variable of β_i, is a *fresh* variable.

   (d) The db-target variable of β_i, $i = 2, \ldots, n$, is the db-output variable of β_{i-1}.

   (e) The db-output variable of the rule head, α, is the same as the db-output variable of the last ordinary subgoal of r, β_n.

□

In the following we show how this algorithm is applied to the transaction definitions for *hire* and *fae* of Examples 5.5 and 5.1, respectively. We use the notation D^i to denote a database variable, so that it is not confused with other variables.

Example 5.12 The transaction definition for *hire* is rewritten as

$$hire(X, Dept, D^0, D^1) \leftarrow qualified(X, D^0) \otimes ins{:}hired(X, Dept, D^0, D^1).$$

Thus, to execute the transaction $?-hire(marketing)$ at initial database, $\mathbf{D}_0$, we simply rewrite this rule as

$$hire(X, marketing, \mathbf{D}_0, D^1) \leftarrow qualified(X, \mathbf{D}_0) \otimes ins{:}hired(X, marketing, \mathbf{D}_0, D^1).$$

[4] For simplicity, assume that each β_i is an ordinary subgoal.

Note that for this program, $hire(marketing)$ is executed only at $\mathbf{D}_0$. Hence, the termination criterion can be *the same as for classical Horn logic.* Bottom-up evaluation can terminate when no more *facts* are produced. □

Example 5.13 The rule definition of fae is rewritten as follows.

$$(r1) \qquad fae(D^0, D^0) \leftarrow \neg emp(X, D^0).$$

$$(r2) \qquad fae(D^0, D^2) \leftarrow emp(X, D^0) \otimes del{:}emp(X, D^0, D^1) \otimes fae(D^1, D^2).$$

Note that fae has two database arguments in the first rule, although it merely queries the database. Its scheme requires, however, two variables, since the second rule indicates that fae is an updating transaction. □

5.3.3 Using Magic Sets for Bottom-Up Evaluation of Horn–$\mathcal{T}_\mathcal{R}$

For general Horn–$\mathcal{T}_\mathcal{R}$-programs, the Magic Sets rule rewriting technique can be used to a) pass variable and database bindings, and b) establish the input sources for updates. Other, specialized techniques for Horn logic can be used as well, with slight modifications.

If the Magic Sets transformation is used, then the input source for an elementary update which is the i-th goal in rule r_j is the supplementary relation $SUP_{j.i-1}$. In general, a projection to the attributes of that update is also needed. The following example shows how the program for fae is rewritten using Magic Sets.

Example 5.14 Given the query goal $? - fae$ at initial database, say $\mathbf{D}_0$, and the rules for fae written as in Example 5.13, the Magic Sets technique creates the following rules

$$m\_fae(\mathbf{D}_0).$$

$$m\_fae(D^1) \leftarrow sup_{2.2}(D^0, D^1).$$

$$sup_{1.0}(D^0) \leftarrow m\_fae(D^0).$$

$$sup_{2.0}(D^0) \leftarrow m\_fae(D^0).$$

$$sup_{2.1}(X, D^0) \leftarrow sup_{2.0}(D^0) \otimes emp(X, D^0).$$

$$sup_{2.2}(D^0, D^1) \leftarrow sup_{2.1}(X, D^0) \otimes del{:}emp(X, D^0, D^1).$$
$$fae(D^0, D^0) \leftarrow sup_{1.0}(D^0) \otimes \neg emp(X, D^0).$$
$$fae(D^0, D^1) \leftarrow sup_{2.2}(D^0, D^1) \otimes fae(D^1, D^2).$$

The input source for the update $del{:}emp(X, D^0, D^1)$ is the relation $SUP_{2.1}$, projected on X, D^0.

This program can be simplified significantly. Ullman [Ull89b] describes an algorithm for simplifying rules produced by the Magic Sets transformation. We can apply those simplifications that *retain the supplementary predicates that are input sources for elementary updates.* The final form of the program, after applying some obvious simplifications, is:

$$m\_fae(\mathbf{D}_0).$$
$$m\_fae(D^1) \leftarrow sup_{2.2}(D^0, D^1).$$
$$sup_{2.1}(X, D^0) \leftarrow m\_fae(D^0) \otimes emp(X, D^0).$$
$$sup_{2.2}(D^0, D^1) \leftarrow sup_{2.1}(X, D^0) \otimes del{:}emp(X, D^0, D^1).$$
$$fae(D^0, D^0) \leftarrow m\_fae(D^0) \otimes \neg emp(X, D^0).$$
$$fae(D^0, D^1) \leftarrow sup_{2.2}(D^0, D^1) \otimes fae(D^1, D^2).$$

□

5.4 Conclusion

We addressed the problem of evaluating definite $\mathcal{T}_{\mathcal{R}}$ programs in a bottom-up fashion. First, we developed a bottom-up evaluation method based on the fixpoint semantics, which uses standard relational technology. Then, we added top-down functionality to bottom-up evaluation, to enable effective handling of updates.

The main characteristic of the proposed bottom-up approach is the bottom-up computation of *all* path-answers for a given transaction. Thus, bottom-up evaluation fails to satisfy our requirement for focusing on one path-answer. As a result, termination in this approach cannot be guaranteed when updates take place. This is not surprising, since the set of path-answers may be infinite when updates occur, even in the function-free case. It is unknown whether additional control can be embedded in bottom-up evaluation so that it terminates when it finds one path-answer. The difficulty lies in determining the completeness of a path-answer.

Example 5.15 Consider again the program of Example 3.3

$$
\begin{array}{ll}
(r1) & a(X) \leftarrow b(X). \\
(r2) & a(X) \leftarrow ins{:}c \otimes d(X). \\
(r3) & d(X) \leftarrow del{:}c \otimes a(X).
\end{array}
$$

$$
\begin{array}{l}
\langle \mathbf{D}_0, \mathbf{D}_1 \rangle ins{:}c \\
\langle \mathbf{D}_1, \mathbf{D}_0 \rangle del{:}c \\
\ldots
\end{array}
$$

and suppose that the initial database is $\mathbf{D}_0$. Given the query $? - a(X)$, bottom-up execution computes all path-answers, $(x, \mathbf{D}_0, \mathbf{D}_1, \mathbf{D}_0, \ldots, \mathbf{D}_1, \mathbf{D}_0)$, such that $b(x)$ is true on $\mathbf{D}_0$. Hence, it fails to terminate if and only if $b(X)$ has at least one answer on $\mathbf{D}_0$. $\square$

Another consequence of its failure to focus on one path-answer is a high demand for space and computation time. Bottom-up processing uses more space than top-down processing even for deterministic transactions, but that difference in both space and time utilization really explodes in the case of non-deterministic transactions. This is because there can be an exponential increase in the number of derived facts, paths and databases. Path maintenance is not expensive compared to database maintenance, since a path is merely a sequence of database descriptors. We expect databases to be stored in terms of differentials, in order to avoid data replication, but we may have to manipulate large differentials, which can be produced by bulk updates. Non-deterministic updates that receive large amounts of input are also an impediment to good performance. This is because they result in both a large number of procedure calls for database changes, and in a large number of facts and paths. Thus, an expensive combination would be a number of non-deterministic choices, each involving several massive updates.

So, can bottom-up evaluation be usefully applied to Horn–$\mathcal{T}_{\mathcal{R}}$ in real domains? The answer is positive. There are two classes of Horn–$\mathcal{T}_{\mathcal{R}}$ programs that could benefit from bottom-up evaluation. First, it would be useful for programs that require few non-deterministic actions, and the number of different path executions is small. The advantage would be gained primarily from the massive computation of queries. Second, it would be useful in situations where *all* path-answers are sought, for instance in planning. Top-down processing would have much lower performance, because it tends to compute one answer at a time.

Research Issues

We have made the assumption that the rules of the initial database remain unchanged during transaction execution. Although this may often be the case, it cannot be taken for granted in general. It is not clear, however, how rule changes can be treated during bottom-up evaluation. Two obvious problems are the modification of the rules for the magic predicates and the validity of a rule for certain databases only. It seems that these problems require run-time solutions, which would add a lot of complication to bottom-up processing. A preventive solution is to evaluate programs which cause rule changes in a top-down fashion.

Parallel manipulation and maintenance of several databases seems to be the tougher implementation problem. We would like to minimize data replication, since only a relatively small amount of data is expected to be affected during the lifetime of a transaction. So, what would be a good combination of storage and access techniques for optimal performance? Does there exist a combination of techniques that uses existing database technology? This is yet another unanswered problem.

Updates are bound to affect only a small portion of the database. Therefore, a relation for a derived predicate in that database will probably remain the same or change slightly. Thus, instead of evaluating a derived predicate from the very beginning for each of its database inputs, it would be advantageous to propagate the changes from stored relations to that derived relation. This is similar to incremental view maintenance (e.g., [GMS93]).

Bottom-up processing works well for queries; it is challenging to investigate whether bottom-up evaluation can be embedded in a top-down evaluation method. Queries do not change database states. In effect, they do not affect the control for rejecting alternative paths. Thus, the process of answering a query can be an independent module. The behaviour of that module is irrelevant, as long as it generates answers correctly. A top-down interpreter could determine what strategy should be adopted for query execution, including bottom-up evaluation. In fact, a different strategy can be used for different queries.

1.

$$
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \end{array}
\bowtie
\begin{array}{lll} \multicolumn{3}{c}{DEL:EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \end{array}
\bowtie
\begin{array}{c} FAE^1_{qa} \\ 2 \end{array}
\overset{\pi_{\$2\$3}}{=}
\begin{array}{cc} \multicolumn{2}{c}{FAE^2_{qa}} \\ 2 & 3 \end{array}
$$

Derived : $\mathbf{D}_{10}$ $\mathbf{D}_{11}$

$$
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \end{array}
$$

2.

$$
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \end{array}
\bowtie
\begin{array}{lll} \multicolumn{3}{c}{DEL:EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
\bowtie
\begin{array}{c} FAE^1_{qa} \\ 2 \end{array}
\overset{\pi_{\$2\$3}}{=}
\begin{array}{cc} \multicolumn{2}{c}{FAE^2_{qa}} \\ 2 & 3 \end{array}
$$

Derived : $\mathbf{D}_2$

3.

$$
\begin{array}{ll} \multicolumn{2}{c}{DOM_{qa}} \\ X & 1 \\ john & \mathbf{D}_0 \\ john & \mathbf{D}_{11} \\ john & \mathbf{D}_2 \\ jack & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ jack & \mathbf{D}_2 \end{array}
-
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \end{array}
\overset{\pi_{\$2}}{=}
\begin{array}{c} FAE^1_{qa} \\ 1 \\ \mathbf{D}_2 \end{array}
$$

4.

$$
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \end{array}
\bowtie
\begin{array}{lll} \multicolumn{3}{c}{DEL:EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
\bowtie
\begin{array}{c} FAE^1_{qa} \\ 2 \\ \mathbf{D}_2 \end{array}
\overset{\pi_{\$2\$3}}{=}
\begin{array}{cc} \multicolumn{2}{c}{FAE^2_{qa}} \\ 1 & 2 \\ \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
$$

5.

$$
\begin{array}{ll} \multicolumn{2}{c}{EMP_{qa}} \\ X & 1 \\ jack & \mathbf{D}_0 \\ john & \mathbf{D}_0 \\ jack & \mathbf{D}_{10} \\ john & \mathbf{D}_{11} \end{array}
\bowtie
\begin{array}{lll} \multicolumn{3}{c}{DEL:EMP_{qa}} \\ X & 1 & 2 \\ john & \mathbf{D}_0 & \mathbf{D}_{10} \\ jack & \mathbf{D}_0 & \mathbf{D}_{11} \\ jack & \mathbf{D}_{10} & \mathbf{D}_2 \\ john & \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
\bowtie
\begin{array}{cc} \multicolumn{2}{c}{FAE^2_{qa}} \\ 2 & 3 \\ \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
\overset{\pi_{\$2\$3\$4}}{=}
\begin{array}{ccc} \multicolumn{3}{c}{FAE^3_{qa}} \\ 1 & 2 & 3 \\ \mathbf{D}_0 & \mathbf{D}_{10} & \mathbf{D}_2 \\ \mathbf{D}_0 & \mathbf{D}_{11} & \mathbf{D}_2 \end{array}
$$

Figure 5.3: Figure for Example 5.7. Semi-naive Evaluation.

Chapter 6

Top-Down Execution of Definite $\mathcal{T}_{\mathcal{R}}$ Programs

As discussed in the previous chapter, bottom-up evaluation does not satisfy our requirement for focusing on one path answer, since it computes the totality of the least fixpoint. In this chapter we develop two theoretical frameworks for *top-down* evaluation methods that satisfy this requirement. In particular, we extend SLD resolution [AvE82] and SLD-AL resolution [Vie87, Vie88, Vie89] to $\mathrm{SLD}^{\mathcal{T}_{\mathcal{R}}}$ and $\mathrm{SLD\text{-}AL}^{\mathcal{T}_{\mathcal{R}}}$ resolution, respectively.

SLD resolution is probably the most widely used framework for query processing in Horn logic, because of its efficiency and its completeness property (e.g., [Llo87]). SLD proof procedures, however, are not *necessarily* complete (e.g., [Llo87]). SLD-AL resolution is one of the approaches proposed as alternatives to SLD resolution [MS81, Loz85, vG86, TS86, Die87, Vie87, Vie88, Vie89]. SLD-AL resolution extends SLD resolution to guarantee completeness for proof procedures in the finite, function-free case, and allows for optimizations that account for the best-known query evaluation methods [Vie88, Vie89]. $\mathrm{SLD\text{-}AL}^{\mathcal{T}_{\mathcal{R}}}$ resolution extends $\mathrm{SLD}^{\mathcal{T}_{\mathcal{R}}}$ resolution in an analogous way.

Section 6.1 defines $\mathrm{SLD}^{\mathcal{T}_{\mathcal{R}}}$ resolution, and proves its soundness and completeness. It also shows that $\mathrm{SLD}^{\mathcal{T}_{\mathcal{R}}}$ proof procedures suffer from similar problems as their SLD counterparts. Section 6.2 proposes SLD-AL$^{\mathcal{T_R}}$ resolution as an extension of SLD-AL and SLD$^{\mathcal{T_R}}$ resolution. It shows that SLD-AL$^{\mathcal{T_R}}$ proof procedures are complete for certain classes of definite $\mathcal{T_R}$ programs.

6.1 SLD$^{\mathcal{T_R}}$ Resolution

Top-down evaluation (also known as *backward chaining*) is based on the idea that a query must be proved to be a logical consequence of a given program. Top-down proof procedures are based on the *resolution inference rule* [Rob65]. SLD resolution is a refinement of this rule, first described by Kowalski [Kow74]. The name "SLD resolution" was given in [AvE82], and stands for Linear resolution with Selection function for Definite clauses.

In this section we define SLD$^{\mathcal{T_R}}$ resolution by extending SLD resolution to account for transaction execution on paths. Our extension easy follows from SLD resolution, in the sense that each result proved for SLD resolution has a counterpart for SLD$^{\mathcal{T_R}}$ resolution. We define SLD$^{\mathcal{T_R}}$ derivations and computed answers, which are the procedural counterpart of correct answers. We prove that every computed answer is correct, and that every correct answer is an instance of a computed answer. This means that SLD$^{\mathcal{T_R}}$ resolution is sound and complete. Finally, we discuss SLD$^{\mathcal{T_R}}$ refutation procedures. Unfortunately, SLD$^{\mathcal{T_R}}$ refutation procedures are shown not to be complete.

6.1.1 SLD$^{\mathcal{T_R}}$ Derivations

As mentioned in Chapter 2, top-down evaluation methods for classical Horn logic regard a clause as a procedure definition and a query goal as a list of procedure calls. To execute a goal (i.e., to execute a procedure call), α, find a clause whose head unifies with α, (i.e., find a procedure definition for α), and execute its body. In Horn–$\mathcal{T_R}$, this reasoning is modified to handle execution paths as follows: To execute a goal, α, *on a database*, find a clause *or elementary transition* whose head unifies with α, and execute its body *on* *that database*; if β is the next clause to be executed, execute β *on the final database of* α.

The following example describes backward chaining for definite $\mathcal{T}_\mathcal{R}$ programs, and introduces *SLD*$^{\mathcal{T}_\mathcal{R}}$ *derivations*. An SLD$^{\mathcal{T}_\mathcal{R}}$ derivation is a possibly infinite sequence of inference steps, taken during a proof that a goal is a logical consequence of a definite $\mathcal{T}_\mathcal{R}$ program. Each inference step corresponds to a procedure call, in the sense mentioned above.

Example 6.1 Consider the transaction definition for fae of Example 5.1.

$$(r1) \qquad fae(Dept) \leftarrow emp(X, Dept) \otimes del{:}emp(X, Dept) \otimes fae(Dept).$$

$$(r2) \qquad fae(Dept) \leftarrow \neg emp(X, Dept).$$

Rule $r1$ can be interpreted in a top-down fashion as follows.

To fire all employees in a department, do the following:

1. Find an employee in that department.

2. Remove that employee from that department.

3. Fire all *remaining* employees.

Rule $r2$ can be interpreted as follows: To fire all employees in a department, check if there is anybody left in that department; if there is nobody left, then the goal succeeds.

Consider the transaction $? - fae(qa)$, which removes all employees who work in the qa department from the initial database, the databases

$\mathbf{D}_0$	$\mathbf{D}_{10}$	$\mathbf{D}_{11}$	$\mathbf{D}_2$
$emp(mike, rd)$	$emp(mike, rd)$	$emp(mike, rd)$	$emp(mike, rd)$
$emp(jack, qa)$	$emp(jack, qa)$	$emp(john, qa)$	
$emp(john, qa)$			

and the following transition base.

$$\langle \mathbf{D}_0, \mathbf{D}_{10} \rangle del{:}emp(john, qa)$$

$\langle \mathbf{D}_0, \mathbf{D}_{11} \rangle del{:}emp(jack, qa)$

$\langle \mathbf{D}_{10}, \mathbf{D}_2 \rangle del{:}emp(jack, qa)$

$\langle \mathbf{D}_{11}, \mathbf{D}_2 \rangle del{:}emp(john, qa)$

...

Assume that $\mathbf{D}_0$ is the initial database. In the following, we treat the facts in the above databases as unit clauses. The goal $fae(qa)$ with initial database $\mathbf{D}_0$ has two alternatives for execution, corresponding to rules r_1 and r_2. This is because $fae(qa)$ *unifies* with the heads of these rules, with substitution $Dept = qa$. Assume that rule $r1$ is chosen for execution. Then, the substitution $Dept = qa$ is applied to the body of $r1$, the initial goal $fae(qa)$ is replaced with the body of rule $r1$, and the new goal

$$emp(X, qa) \otimes del{:}emp(X, qa) \otimes fae(qa)$$

is derived to be evaluated on $\mathbf{D}_0$. This goal is a serial conjunction, so it evaluated from left to right. The subgoal $emp(X, qa)$ unifies with the unit clauses $emp(john, qa)$ and $emp(jack, qa)$ in $\mathbf{D}_0$, with substitutions $X = john$ and $X = jack$, respectively. Let us choose one of these, say $X = john$. By applying this substitution to the rest of the body, a new goal,

$$del{:}emp(john, qa) \otimes fae(qa),$$

is derived to be evaluated on $\mathbf{D}_0$. The subgoal $del{:}emp(john, qa)$ unifies with the head of the elementary transition

$$\langle \mathbf{D}_0, \mathbf{D}_{10} \rangle del{:}emp(john, qa),$$

which says that $del{:}emp(john, qa)$ transforms $\mathbf{D}_0$ into $\mathbf{D}_{10}$. Thus, the evaluation of subgoal $del{:}emp(john, qa)$ derives a new database, i.e., $\mathbf{D}_{10}$, and the goal

$$fae(qa)$$

to be evaluated on $\mathbf{D}_{10}$. Note that this goal is the same as the initial goal, but it is evaluated on a different database. Repeating the above process with database $\mathbf{D}_{10}$, we derive the goal $fae(qa)$ on $\mathbf{D}_2$. This goal fails if $r1$ is chosen again, because there is no tuple in $\mathbf{D}_2$ of the form $emp(name, qa)$. It succeeds, however, if rule $r2$ is chosen for execution, leaving $\mathbf{D}_2$ intact. We say that the above sequence of steps constructs an *SLD*$^{\mathcal{T}_{\mathcal{R}}}$ *derivation* for $fae(qa)$ on the path $\mathbf{D}_0, \mathbf{D}_{10}, \mathbf{D}_2$. This path is a possible execution path. □

In the following we formalize the above discussion. We first define SLD$^{\mathcal{T}_{\mathcal{R}}}$ derivations, extending the definition of SLD derivations in [AvE82].

Definition 6.1 (*SLD*$^{\mathcal{T}_{\mathcal{R}}}$ *derivations)* Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D}_0)$ be a definite $\mathcal{T}_{\mathcal{R}}$-program, N a negative clause, and $\mathbf{D}_0, \mathbf{D}_1, \ldots$ a database sequence. An SLD$^{\mathcal{T}_{\mathcal{R}}}$ derivation of N on the path $\mathbf{D}_0, \mathbf{D}_1, \ldots$ consists of a sequence $N_1, N_2, \ldots$ of negative clauses, a sequence $d_1, d_2, \ldots$ of clauses or elementary transitions (called the *input axioms* of the derivation), and a sequence of $\theta_1, \theta_2, \ldots$ of variable substitutions.

We call the database sequence $\mathbf{D}_0, \mathbf{D}_1, \ldots$ the *path of the derivation*. Database $\mathbf{D}_{j+1}$ in the path of the derivation is said to be *derived* from database $\mathbf{D}_j$. Each N_i is associated with a database, $\mathbf{D}_j$, in the path of the derivation. Intuitively, N_i represents a transaction to be executed on the database that is associated with. We thus say that N_i *is evaluated* on the database $\mathbf{D}_j$. In particular, N_1 is evaluated on the initial database, $\mathbf{D}_0$. If N_i is non-empty, then the clause N_{i+1} is *derived* from N_i and d_i, using substitution θ_i and the *selected atom* of N_i. The latter is an atom of N_i selected by using some function, called *selection function* (or *computation rule*), as follows:

Let $N_i \equiv \leftarrow \alpha$.

Set *selected* to α.

While *selected* is not an atom do

  Let *selected* $\equiv [\ \alpha_1 \ \& \ldots \& \ \alpha_n\ ] \otimes rest$,

    where each α_i is an atom or a parenthesized formula.

Set *selected* to α_i, for some i, $1 \leq i \leq n$, using a selection function.

end.

Clauses and databases are derived as follows. Let

$$N_i \equiv \leftarrow [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest,$$

where α_s is the selected atom of N_i, and let N_i be evaluated on database $\mathbf{D}_j$. N_{i+1} can be derived in two ways.

1. Let $d_i \equiv \alpha \leftarrow \beta$ be a variant of a clause in $\mathbf{D}_j$ or $\mathbf{P}$ which has no variables in common with N_i, and such that α and α_s are unifiable with most general unifier (mgu) θ. Then N_{i+1} is

$$N_{i+1} \equiv \leftarrow (\ [\ \ldots \&\ \alpha_{s-1}\ \&\ [\ \beta\ ]\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta.$$

   N_{i+1} is evaluated on $\mathbf{D}_j$, and it is called the *resolvent* of N_i and d_i. For each atom β_j in β, $\beta_j\theta$ is said to be *introduced* in N_{i+1} by d_i. For each atom α_j in N_i, other than the selected atom, $\alpha_j\theta$ is said to be *derived* from α.

2. Let $d_i \equiv \langle \mathbf{D}, \mathbf{D}' \rangle \alpha$ be an elementary transition in P, such that $\mathbf{D} \equiv \mathbf{D}_j$, and α, α_s are unifiable with mgu θ. Then,

$$\mathbf{D}_{j+1} \equiv \mathbf{D}'$$

   and

$$N_{i+1} \equiv \leftarrow (\ [\ \ldots \&\ \alpha_{s-1}\ \&\ \alpha_{s+1}\ \&\ldots\ ] \otimes rest\ )\theta.$$

   N_{i+1} is evaluated on $\mathbf{D}_{j+1}$.

$\Box$

An *SLD*$^{\mathcal{T}_\mathcal{R}}$ *refutation* of N is a *finite* SLD$^{\mathcal{T}_\mathcal{R}}$ derivation of N which has the empty clause, $\Box$, as the last clause in the derivation. If $N_n = \Box$ then we say that the refutation has length n. An *unrestricted* $SLD^{\mathcal{T_R}}$ *refutation* is an SLD$^{\mathcal{T_R}}$ refutation which does not necessarily use substitutions that are mgu's. An $SLD^{\mathcal{T_R}}$ *refutation via* R is a refutation generated by using computation rule R. SLD$^{\mathcal{T_R}}$ derivations can be finite or infinite. A *successful* derivation is a refutation. A *failed* derivation is one that ends in a non-empty negative clause whose head does not unify with the head of any program clause or elementary transition.

If $\theta_1, \theta_2, \ldots$ is the sequence of substitutions of a refutation, then their composition provides bindings to the free variables of the initial goal of the refutation and is called the *answer substitution* of the refutation. The answer substitution, θ, and the path, Δ, of a refutation compose a partial answer, (θ, Δ). A *computed partial path-answer* (θ', Δ) for N is the partial answer obtained by restricting θ to the variables of N, and it is the procedural counterpart of the correct answer. SLD$^{\mathcal{T_R}}$ resolution is *sound* if every computed partial answer for N is a correct answer for N. The procedural counterpart of the least Herbrand model is the *success path structure*, defined as follows.

Definition 6.2 (*Success Path Structure)* Let P be a definite $\mathcal{T_R}$ program. The success path structure of P is a Herbrand path structure $\langle U_{\mathcal{L}},\ I_{\mathcal{F}\mathcal{L}},\ N,\ I_{path}\rangle$, where I_{path} is defined as follows: For each possible database sequence $\mathbf{D}_0, \ldots, \mathbf{D}_n$ in P,
$\alpha \in I_{path}(<s^{\mathbf{D}_0}, \ldots, s^{\mathbf{D}_n}>)$ if and only if $\leftarrow \alpha$ has an SLD$^{\mathcal{T_R}}$ refutation on $\mathbf{D}_0, \ldots, \mathbf{D}_n$, where $s^{\mathbf{D}_i}$ means that state s corresponds to database $\mathbf{D}_i$. $\Box$

Note that if a goal, N, is a query, then an SLD$^{\mathcal{T_R}}$ derivation of N is an SLD derivation. Since SLD resolution is sound and complete, the mapping, I_{path}, of the success path structure satisfies the requirement that $I_{path}(<s>) \in s$, i.e., it is a valid path structure. As we will see, the success path structure of P is equal to the least Herbrand model of P, $\mathbf{M}_P$.

6.1.2 Soundness of SLD$^{\mathcal{T_R}}$ resolution

The following lemma in [BK93] lists some properties of executional entailment, which we use to prove the soundness of SLD$^{\mathcal{T_R}}$ resolution.

Lemma 6.1 (*Basic Properties of Executional Entailment* [BK93]) For any transition base $\mathcal{B}$, any transaction base $\mathbf{P}$, any database sequence $\mathbf{D}_0, \ldots, \mathbf{D}_n$, and any transaction formulae α and β, the following statements are all true:

1. If $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \wedge \beta$.
2. If $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_i \models \alpha$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_i, \ldots, \mathbf{D}_n \models \beta$ then
   (a) $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \otimes \beta$.
   (b) $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha \ \& \ \beta$
   (c) $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta \ \& \ \alpha$.
3. If $\alpha \leftarrow \beta$ is in $\mathbf{P}$ and $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \beta$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$.
4. If $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$ is in $\mathcal{B}$, then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha$.
5. If $\alpha \leftarrow$ is in $\mathbf{P}$, then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_n \models \alpha$, for all possible database sequences $\mathbf{D}_0, \ldots, \mathbf{D}_n$
6. If $\mathbf{D}_0 \models^c \psi$ then $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \psi$, where ψ is a first-order formula, and $\models^c$ denotes classical entailment.

□

Theorem 6.1 (*Soundness of SLD$^{\mathcal{T}_{\mathcal{R}}}$ Resolution)* Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite $\mathcal{T}_{\mathcal{R}}$-program and N a definite goal. Then every computed answer for N is a correct answer for N.

Proof Let N be the goal

$$\leftarrow [\ \ldots \alpha_{s-1} \ \& \ \alpha_s \ \& \ \alpha_{s+1} \ldots\ ] \otimes rest,$$

and consider a refutation of N on $\mathbf{D}_0, \ldots, \mathbf{D}_m$ of length n, which uses substitutions $\theta_1, \ldots, \theta_n$. We have to show that $(\ [\ \ldots \alpha_{s-1} \ \& \ \alpha_s \ \& \ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n$ is a logical consequence of P on the path $<s^{\mathbf{D}_0}, \ldots, s^{\mathbf{D}_m}>$, i.e., that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

We prove the result by induction on the length of the refutation, using Lemma 6.1. All statements refer to that lemma.

Basis. Assume that $n = 1$. This means that N is of the form $\leftarrow \alpha_1$. Let also d be the first input axiom and α be its head. Then,

$$\alpha\theta_1 = \alpha_1\theta_1. \tag{6.1}$$

We distinguish the following cases with respect to d.

- d is a unit clause in $\mathbf{D}_0$, i.e., $d \equiv \alpha \leftarrow$. Then $m = 0$. By Statement 6,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \models \alpha\theta_1.$$

- d is a unit clause in $\mathbf{P}$, i.e., $d \equiv \alpha \leftarrow$. Then, the database sequence can be any sequence of databases that appear in $\mathcal{B}$. Since d is a unit clause in $\mathbf{P}$, by Statement 5,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models \alpha\theta_1.$$

- d is an elementary transition in $\mathcal{B}$, i.e., $d \equiv \langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$. Then, $m = 1$. By Statement 4,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \models \alpha\theta_1.$$

In all three cases, by equation 6.1, we have that $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models \alpha_1\theta_1$.

Induction step. Assume now that the result holds for computed answers that correspond to refutations of length $n-1$. Let d be the first input axiom and α be its head. Let also α_s be the selected atom of N. Then

$$\alpha\theta_1 \ldots \theta_n = \alpha_s\theta_1 \ldots \theta_n \tag{6.2}$$

We distinguish the following cases with respect to d.

- d is a clause in $\mathbf{D}_0$, i.e., $d \equiv \alpha \leftarrow \beta$. By the induction hypothesis,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ [\ \beta\ ]\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

Thus, $\beta\theta_1 \ldots \theta_n$ must be a logical consequence of P on $<s^{\mathbf{D}_j}>$, for some $\mathbf{D}_j$ in the database sequence, i.e.,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_j \models \beta\theta_1 \ldots \theta_n.$$

By Statement 3 and equation 6.2,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_j \models \alpha_s\theta_1 \ldots \theta_n.$$

By replacing β by α_s, we have that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

- d is a clause in $\mathbf{P}$, i.e., $d \equiv \alpha \leftarrow \beta$. By the induction hypothesis,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ [\ \beta\ ]\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

Thus, it must be the case that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_j, \ldots, \mathbf{D}_{j+k} \models \beta\theta_1 \ldots \theta_n,$$

for a some subsequence $\mathbf{D}_j, \ldots, \mathbf{D}_{j+k}$ of the database sequence. Hence, by Statement 3 and equation 6.2,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_j, \ldots, \mathbf{D}_{j+k} \models \alpha_s\theta_1 \ldots \theta_n.$$

By replacing β by α_s, we have that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

- d is an elementary transition in $\mathcal{B}$, i.e., $d \equiv \langle \mathbf{D}_0, \mathbf{D}_1 \rangle \alpha$. By the induction hypothesis,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_1, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

By Statement 4 and equation 6.2,

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_1, \mathbf{D}_2 \models \alpha_s \theta_1 \ldots \theta_n.$$

By applying Statement 2, we have that

$$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \ldots, \mathbf{D}_m \models (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest\ )\theta_1 \ldots \theta_n.$$

□

6.1.3 Completeness of SLD$^{\mathcal{T}_\mathcal{R}}$ resolution

We prove the completeness of SLD$^{\mathcal{T}_\mathcal{R}}$ resolution based on the respective proof for of SLD resolution given in [Llo87].

Lemma 6.2 (*Mgu Lemma for SLD$^{\mathcal{T}_\mathcal{R}}$ resolution*) Let P be a definite $\mathcal{T}_\mathcal{R}$ program and N a definite goal. Suppose that N has an unrestricted SLD$^{\mathcal{T}_\mathcal{R}}$ refutation on a path, π. Then N has an SLD$^{\mathcal{T}_\mathcal{R}}$ refutation on π of the same length, such that if $\theta_1, \ldots, \theta_n$ are the unifiers from the unrestricted SLD$^{\mathcal{T}_\mathcal{R}}$ refutation and $\theta'_1, \ldots, \theta'_n$ are the mgu's from the SLD$^{\mathcal{T}_\mathcal{R}}$ refutation, then there exists a substitution γ such that $\theta_1, \ldots, \theta_n = \theta'_1, \ldots, \theta'_n\gamma$.
Proof Noting that the most general unifier is the only unifier for an atom and a head of an elementary transition, the result is proved as in [Llo87]. □

Lemma 6.3 (*Lifting Lemma for SLD$^{\mathcal{T}_{\mathcal{R}}}$ resolution*) Let P be a definite $\mathcal{T}_{\mathcal{R}}$ program, N a definite goal, and θ a substitution. Suppose there exists an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of $\leftarrow N\theta$ on a path, π. Then, there exists an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of $\leftarrow N$ on π of the same length, such that if $\theta_1, \ldots, \theta_n$ are the mgu's from the SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of $\leftarrow N\theta$, and $\theta'_1, \ldots, \theta'_n$ are the mgu's from the SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of $\leftarrow N$, then there exists a substitution, γ, such that $\theta\theta_1, \ldots, \theta_n = \theta'_1, \ldots, \theta'_n\gamma$.

Proof As in [Llo87]. $\Box$

Theorem 6.2 The success path structure of a definite program P is equal to its least Herbrand model.

Proof Let I_{path} and $I^{M_P}_{path}$ be the mappings of the success path structure and the least Herbrand model of P, respectively. It suffices to prove that $I^{M_P}_{path}(\pi)$ is included in $I_{path}(\pi)$, for each path π in M_P, and vise versa.

- $I_{path}(\pi) \subseteq I^{M_P}_{path}(\pi)$

  Let α be an atom in $I_{path}(\pi)$. Then, $\leftarrow \alpha$ has a refutation on π. By the soundness of SLD$^{\mathcal{T}_{\mathcal{R}}}$ resolution, α is a logical consequence of P on π. Thus, α is in $I^{M_P}_{path}(\pi)$.

- $I^{M_P}_{path}(\pi) \subseteq I_{path}(\pi)$

  Let α be an atom in $I^{M_P}_{path}(\pi)$. Then, by Theorem 4.2, $\alpha \in I^{T_{\mathcal{B},\mathrm{P}}\uparrow n}_{path}(\pi)$, for some integer n. We prove by induction on n that $\alpha \in I^{T_{\mathcal{B},\mathrm{P}}\uparrow n}_{path}(\pi)$ implies that $\leftarrow \alpha$ has a refutation on π, and α is therefore in $I_{path}(\pi)$.

  Basis. Assume that $n = 1$. Then α must be (a) a ground instance of a unit clause in $\mathbf{P}$, (b) a ground instance of an elementary transition, or (c) π is $<s>$, for some state s that corresponds to a database, $\mathbf{D}$, mentioned in the transition base, and α is a logical consequence of the database $\mathbf{D}$. In the first two cases, it is obvious that $\leftarrow \alpha$ has an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation on π. In third case, there is an SLD refutation of α (by the completeness of SLD resolution). Thus, there is an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of α on $\mathbf{D}$ (since every SLD derivation is an SLD$^{\mathcal{T}_{\mathcal{R}}}$ derivation).

Induction step. Assume that the result holds for $n-1$. Let $\alpha \in I_{path}^{T_{\mathcal{B},\mathbf{P}}\uparrow n}(\pi)$. Then α must be a derived atom. By the definition of $T_{\mathcal{B},\mathbf{P}}$, there exists a ground instance of a clause, $\beta \leftarrow \gamma$, in $\mathbf{P}$ such that $\alpha = \beta\theta$ and $I_{path}^{T_{\mathcal{B},\mathbf{P}}(\pi)\uparrow(n-1)}(\pi) \models \gamma\theta$. Let $\beta_1, \ldots, \beta_k$ be all the atoms appearing in γ. Then, for some order, $\beta_{i_1}, \ldots, \beta_{i_k}$, of these atoms,[1] there exist paths $\pi_1, \ldots, \pi_k$, such that

$$I_{path}^{T_{\mathcal{B},\mathrm{P}}(\pi)\uparrow(n-1)}(\pi_1) \models \beta_{i_1}\theta$$

$$\vdots$$

$$I_{path}^{T_{\mathcal{B},\mathrm{P}}(\pi)\uparrow(n-1)}(\pi_k) \models \beta_{i_k}\theta \text{ and}$$

$$\pi = \pi_1 \circ \ldots \circ \pi_k.$$

By the induction hypothesis, β_{i_j} has a refutation on π_j, $j = 1, \ldots, k$. Since each β_{i_j} is ground, these refutations can be combined into a refutation for $\leftarrow \gamma\theta$ on π. Thus, $\leftarrow \alpha$ has an unrestricted refutation on π, with substitution θ. A refutation for $\leftarrow \alpha$ on π is obtained by applying the mgu lemma.

□

Lemma 6.4 Let P be a definite $\mathcal{T_R}$ program and α an atom, such that $\forall(\alpha)$ is a logical consequence of P on a path, π. Then, there exists an $\mathrm{SLD}^{\mathcal{T_R}}$ refutation of $\leftarrow \alpha$ on π, such that the substitution of the computed answer is the identity.

Proof As in [Llo87]. □

The major completeness result follows.

[1] If γ is a serial conjunction, this order is from left to right. If shuffle operators appear in γ, the order might be different.

Theorem 6.3 (*Completeness of $SLD^{\mathcal{T_R}}$ resolution*) Let P be a definite $\mathcal{T_R}$ program and N a definite goal. For every correct partial answer, (θ, π), for N, there exists a computed answer, (σ, π), for N and a substitution, γ, such that $\theta = \sigma\gamma$.

Proof Let N be $\leftarrow \alpha$. Since (θ, π) is correct, $\forall(\alpha\theta)$ is a logical consequence of P on π. Let $\beta_1, \ldots, \beta_k$ be all the atoms appearing in α. Then, for some order, $\beta_{i_1}, \ldots, \beta_{i_k}$, of these atoms, there exist some paths, $\pi_1, \ldots, \pi_k$, such that

$$\mathbf{M}_P, \pi_1 \models \beta_{i_1}\theta$$

$$\vdots$$

$$\mathbf{M}_P, \pi_k \models \beta_{i_k}\theta \text{ and}$$

$$\pi = \pi_1 \circ \ldots \circ \pi_k.$$

By Lemma 6.4, there exists a refutation of $\leftarrow \beta_{i_j}\theta$ on π_j, such that the computed answer is the identity, $i = 1, \ldots, k$. We can combine these refutations into a refutation of $\leftarrow \alpha\theta$ on π, such that the substitution of the computed answer is the identity. Let $\theta_1, \ldots, \theta_n$ be the sequence of mgu's of this refutation of $\leftarrow \alpha\theta$ on π. Then, $\alpha\theta\theta_1, \ldots, \theta_n = \alpha\theta$. By the lifting lemma, there exists a refutation of $\leftarrow \alpha$ on π with mgu's $\theta'_1 \ldots, \theta'_n$, such that $\theta\theta_1, \ldots, \theta_n = \theta'_1 \ldots, \theta'_n\gamma$, for some substitution γ. Let σ be $\theta'_1, \ldots, \theta'_n$ restricted to the variables in $\leftarrow \alpha$. Then, (σ, π) is a computed answer for $\leftarrow \alpha$, and $\theta = \sigma\gamma$, where γ is an appropriate restriction of γ'. $\square$

6.1.4 SLD$^{\mathcal{T_R}}$ Refutation Procedures

There are several strategies that an interpreter for definite $\mathcal{T_R}$ programs may adopt in its search for SLD$^{\mathcal{T_R}}$ refutations. We call these strategies *SLD$^{\mathcal{T_R}}$ refutation procedures.* An SLD$^{\mathcal{T_R}}$ refutation procedure searches for refutations by constructing SLD$^{\mathcal{T_R}}$ derivations. The totality of derivations constructed by a refutation procedure is called the *search space* of the procedure. The search space we consider is called *SLD$^{\mathcal{T_R}}$ tree*, and it is an extension of SLD tree. A refutation procedure is characterized by a) *the selection* *function* for the construction of SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees, and b) the *search strategy* it uses for finding successful branches of the tree. In the following we define SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees and introduce search strategies.

SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees

According to the mgu lemma, it suffices to include in the search space only derivations that use the most general unifier. We have to consider, however, alternatives of selection functions for the construction of the search space. This is because (as discussed in Chapter 5) a list of goals, $t_1 \,\&\ldots\&\, t_n$, might have a number of different executions. One selection function might lead to a failed derivation, while another leads to a successful one. In SLD resolution, refutation procedures are independent of the selection function, i.e., any choice of selection function leads to a complete search space [Llo87]. This is true, because queries do not have side effects. This property reduces the search space, and allows SLD refutation procedures to focus on the search strategy. For these reasons, we would like SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation procedures to have this property.

This is not hard to achieve. Recall from Chapter 5, that any program can be rewritten to satisfy the free execution order property, so that bottom-up evaluation of a list of goals, $t_1 \,\&\ldots\&\, t_n$, is independent of the chosen order of these goals. The free execution order property required in bottom-up evaluation is the counterpart of the independence of the selection function in top-down evaluation. That is, any program can be rewritten, so that any choice of selection function leads to complete search spaces. Therefore, we set the following requirement.

Requirement 6.1 (*Independence of the selection function*) Let P be a definite $\mathcal{T}_{\mathcal{R}}$ program and N a definite goal. Suppose that there is an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation with computed answer (θ, Δ). Then, for any selection function, R, there exists an SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation of N via R with R-computed answer (θ', Δ), such that $N\theta'$ is a variant of $N\theta$. $\square$

Different computation rules produce SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees which might differ substantially in size. Thus, the choice of computation rule is important, as far as performance is concerned.

Paths in an $\text{SLD}^{\mathcal{T_R}}$ tree are $\text{SLD}^{\mathcal{T_R}}$ derivations, and every path to the empty clause is a refutation. The latter are called *success* branches. Branches that correspond to failed and infinite derivations are called *failure* and *infinite* branches, respectively. In the absence of updates, an $\text{SLD}^{\mathcal{T_R}}$ tree is an SLD tree.

Definition 6.3 (*$SLD^{\mathcal{T_R}}$ tree)* Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D}_0)$ be a definite program and N a definite goal. An $\text{SLD}^{\mathcal{T_R}}$ tree for N is a tree rooted at N, whose nodes are either negative or empty clauses. Each arc is labeled with a substitution and possibly a database, called the *database label* of the arc. The database labels of consecutive arcs build a (possibly empty) database sequence. Each node, N_i, is associated with a database, $\mathbf{D}_j$, which is the database label of the first arc in the path from that node to the root. We say that N_i is *evaluated* on $\mathbf{D}_j$. The root is evaluated on the initial database, $\mathbf{D}_0$.

Let

$$N_i \equiv\leftarrow [\ \ldots \alpha_{s-1}\ \&\ \alpha_s\ \&\ \alpha_{s+1} \ldots\ ] \otimes rest, \quad k \geq 1,$$

be any node, evaluated on a database, $\mathbf{D}_j$, and α_s be the selected atom. N_i has a child for:

- Every definite clause $\alpha \leftarrow \beta$ in $\mathbf{P}$ or $\mathbf{D}_j$, such that α_s and α are unifiable using the most general unifier, say θ. The descendant is

  $$N_{i+1} \equiv\leftarrow (\ [\ \ldots \&\ \alpha_{s-1}\ \&\ [\ \beta\ ]\ \&\ \alpha_{s+1}\ \& \ldots\ ] \otimes rest\ )\theta,$$

  and the arc from N_i to N_{i+1} is labeled by θ. For each atom β_j in β, $\beta_j\theta$ is said to be *introduced* in N_{i+1} by $\alpha \leftarrow \beta$. For each atom α_j in N_i, other than the selected atom, $\alpha_j\theta$ is said to be *derived* from α.

- Every elementary transition $\langle \mathbf{D}, \mathbf{D}' \rangle \alpha$ in P, such that α_s and α are unifiable using the mgu, say θ, and $\mathbf{D} \equiv \mathbf{D}_j$. The descendant is

  $$N_{i+1} \equiv\leftarrow (\ [\ \ldots \alpha_{s-1}\ \&\ \alpha_{s+1}\ \& \ldots\ ] \otimes rest\ )\theta,$$

and the arc from N_i to N_{i+1} is labeled by θ and $\mathbf{D}'$.

□

The *descendance relationship* for tree nodes is the same as for SLD resolution.

Definition 6.4 (*The descendance and ancestor relationship)* A node, N, is a *direct descendant* of a node, N', if the subgoal selected at N was introduced at N'. The *descendance relationship* is the transitive closure of the direct descendance relationship. N' is an *ancestor* of N, if N is a descendant of N'. □

To obtain a computed answer (i.e., a possible execution), one can follow a successful branch, starting from the root. The composition of the database labels is the execution path, and the composition of the substitutions restricted on the variables of the root is the answer substitution.

Example 6.2 Consider again the transaction definition for fae in Example 6.1, and the transaction $? - fae(qa)$. Part of the SLD$^{\mathcal{T}_{\mathcal{R}}}$ tree for this transaction is shown in Figure 6.2. For clarity, we do not show the failure branches. This is an example of a *finite* SLD$^{\mathcal{T}_{\mathcal{R}}}$ tree. One can clearly see the two successful derivations that correspond to the two possible executions of $? - fae(qa)$. By following the left successful branch, we obtain the execution path $\mathbf{D}_0, \mathbf{D}_{10}, \mathbf{D}_2$, while the composition of the substitutions restricted on the variables of the root (which has no variables) is the identity substitution. Similarly, the right successful branch gives the execution path $\mathbf{D}_0, \mathbf{D}_{11}, \mathbf{D}_2$. □

Search Strategies

Although the completeness of SLD$^{\mathcal{T}_{\mathcal{R}}}$ resolution ensures that every answer to a query can be found by following some computation path in an SLD$^{\mathcal{T}_{\mathcal{R}}}$ tree, SLD$^{\mathcal{T}_{\mathcal{R}}}$ refutation procedures are not necessarily complete or terminating. This is not surprising, since SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees are *infinite* search spaces. Infiniteness comes into the form of an *infinite number* of branches, or *infinite* branches. Transactions with infinite path-answers or

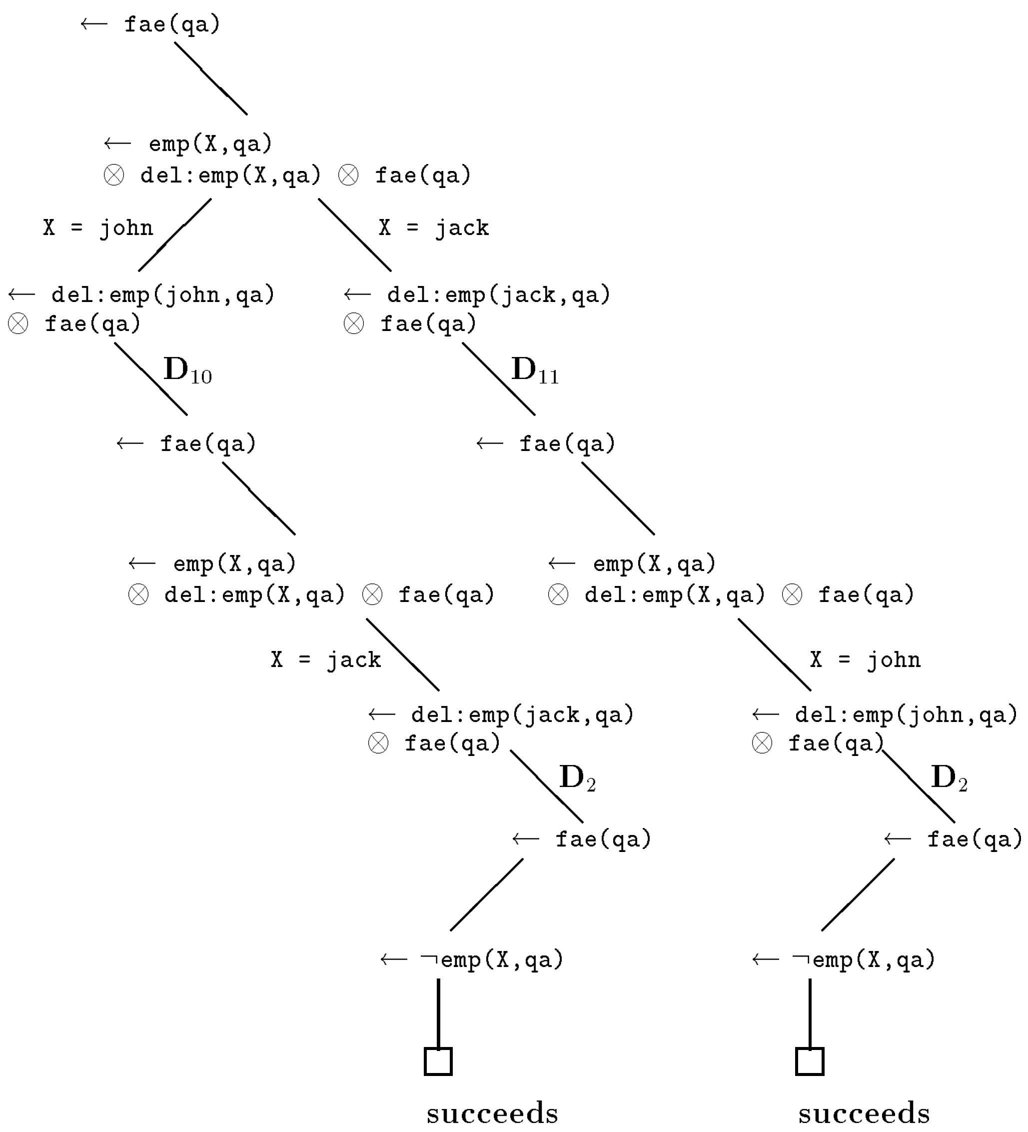

Figure 6.1: $\mathrm{SLD}^{\mathcal{T}_{\mathcal{R}}}$ tree for Example 6.2. (Failure branches are not shown.)

infinitely many path-answers give rise to the former form of infiniteness. Infinite branches are independent of the number or the nature of path-answers, i.e., they may arise even if a transaction has a finite number of finite path-answers. This is particularly true in the presence of cyclic data and updates, and unfortunately, it is true even for queries. This is the reason that SLD refutation procedures are not necessarily complete. Prolog [Bra91], for instance, might get caught in a infinite branch of an SLD tree, and fail to terminate or return all answers to a given query. Experienced Prolog programmers know how to write programs to prevent Prolog's interpreter from getting stuck in an infinite branch, but this is not always possible [Llo87, Ull89b]. In Horn–$\mathcal{T_R}$, this is particularly true for updating transactions, because it is the nature of updates that does not allow much flexibility in writing a program. Thus, for some programs, *any* top-down search procedure based on $\mathrm{SLD}^{\mathcal{T_R}}$ resolution fails to terminate.

The following example discusses the role of the choice of selection function and cyclic data in the creation of infinite branches.

Example 6.3 Let us consider the following two Horn programs that solve the reachability problem in graphs.

$$
\begin{aligned}
P_1: \quad & r(X,Y) \leftarrow r(X,Z) \;\&\; e(Z,Y). \\
& r(X,Y) \leftarrow e(X,Y).
\end{aligned}
$$

$$
\begin{aligned}
P_2: \quad & r(X,Y) \leftarrow e(X,Z) \;\&\; r(Z,Y). \\
& r(X,Y) \leftarrow e(X,Y).
\end{aligned}
$$

and the goal $\leftarrow r(1,X)$. Figure 6.2 depicts part of an $\mathrm{SLD}^{\mathcal{T_R}}$ tree for $P_1 \cup \{\leftarrow r(1,Y)\}$ with the selected atoms underlined. Apparently, any search procedure gets trapped in the infinite branch of the tree and fails to terminate. Even worse, a *depth-first procedure* that follows blindly the rightmost branch fails to return *any answer*. A *breadth-first* procedure is guaranteed to return all answers to the top goal eventually, but it tends to compute the same local answers repeatedly. Furthermore, breadth-first search is undesirable due to its large demand for memory [Ull89b].

Assume now that we construct the SLD$^{\mathcal{T}_\mathcal{R}}$ tree by using the e-subgoals for selected atoms. Then it is not hard to see that the tree is finite if and only if the data graph is *acyclic*. Suppose, for example, that the data graph is $\{e(1,2), e(2,1)\}$. Then, as Figure 6.3 shows, the corresponding SLD$^{\mathcal{T}_\mathcal{R}}$ tree has an infinitely periodic structure that repeats the subtrees rooted at goals $\leftarrow r(1,2)$ and $\leftarrow r(1,1)$. Furthermore, there is some added cost for evaluating the e-goals, because all their variables are free.

In the case that the data graph has no cycles, program P_2 appears to be a good solution for answering the given goal. It does not, however, avoid the aforementioned problems given the goal $P_2 \cup \{\leftarrow r(X,1)\}$ (Figure 6.2). $\Box$

The following example shows how updates limit flexibility in writing programs.

Example 6.4 Consider now the following modifications of programs P_1 and P_2

$$\begin{aligned} \bar{P}_1 : \quad & ir(X,Y) \leftarrow ir(X,Z) \;\&\; e(Z,Y). \\ & ir(X,Y) \leftarrow e(X,Y) \otimes ins{:}arc(X,Y). \end{aligned}$$

$$\begin{aligned} \bar{P}_2 : \quad & ir(X,Y) \leftarrow e(X,Z) \;\&\; ir(Z,Y). \\ & ir(X,Y) \leftarrow e(X,Y) \otimes ins{:}arc(X,Y). \end{aligned}$$

and the transaction $?-ir(1,Y)$. Path-answers for its execution with respect to $\bar{P}_1$ are of the form $(\{y\},\ \mathbf{D}, \mathbf{D} \cup \{arc(1,x)\})$, where x is such that $e(1,x)$ is true, and y is reachable from x. The meaning of program $\bar{P}_1$, however, is different from that of $\bar{P}_2$. A path-answer for the execution of the given transaction with respect to $\bar{P}_2$ is of the form $(\{y\},\ \mathbf{D}, \mathbf{D} \cup \{arc(x,y)\})$, where x is reachable from 1, and $e(x,y)$ is true. Thus, the set of answer substitutions of each path-answer is singleton.

$\bar{P}_1$ and $\bar{P}_2$ face the same problem of infiniteness as P_1 and P_2. We also see that updates limit rewriting flexibility. This is in the nature of updates. For instance, "load

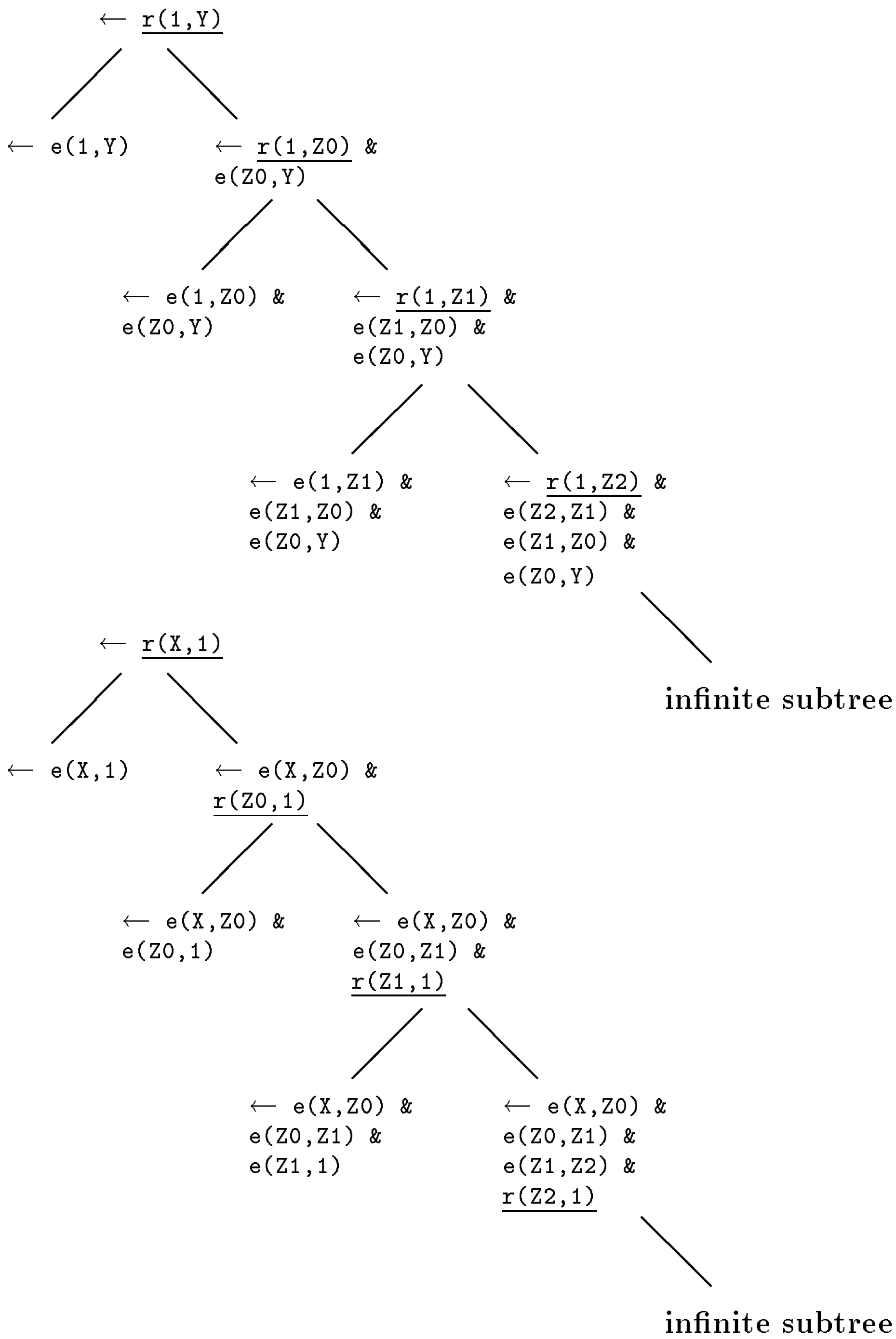

Figure 6.2: $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ trees for Example 6.3.

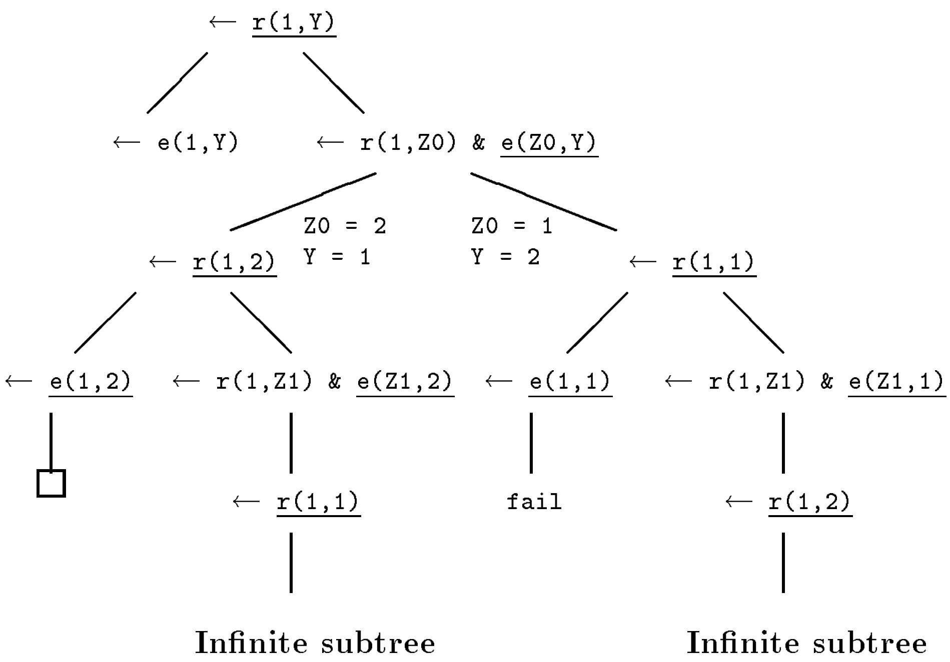

Figure 6.3: Infinite SLD$^{\mathcal{T}_\mathcal{R}}$ tree due to cyclic data.

the gun, then pull the trigger" is different from "pull the trigger, then load the gun", though the syntactic difference is small. Seemingly minor changes of a program might alter its meaning completely. □

The following example discusses the role of updates in the creation of infinite branches.

Example 6.5 Consider again the program of Example 3.3. Then, there is only one SLD$^{\mathcal{T}_\mathcal{R}}$ tree for this program, starting from initial database **D**, with an infinite branch that repeats the goals $a(X)$ and $d(X)$ at databases **D** and **D′**, respectively (Figure 6.4). This branch prevents *any* top-down search procedure from termination. This is true even if $b(X)$ has no answer at **D**. Remember that in this case bottom-up evaluation terminates. □

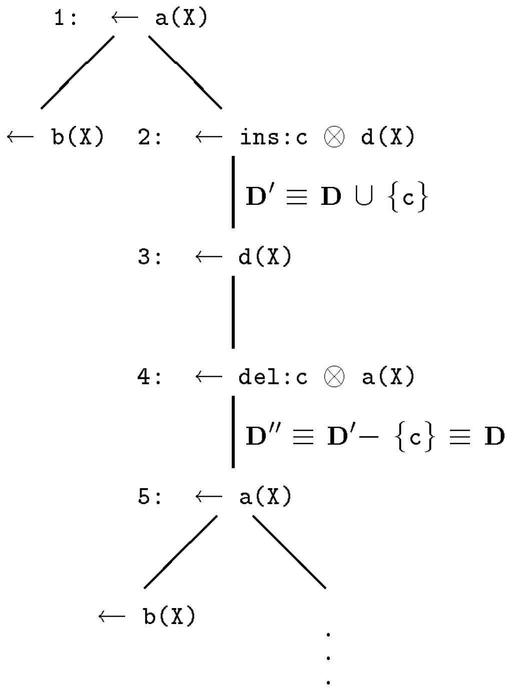

Figure 6.4: Figure for Example 6.9.

6.2 SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ Resolution

6.2.1 Introduction

SLD$^{\mathcal{T}_{\mathcal{R}}}$ resolution has two nice properties which are desirable for transaction execution: It constructs complete search spaces, and it is efficient. On the other hand, the non-finiteness of SLD$^{\mathcal{T}_{\mathcal{R}}}$ tree branches is a source of major problems. The construction of SLD$^{\mathcal{T}_{\mathcal{R}}}$ trees is naive in that the subgoals to be resolved are selected blindly. The processing of similar subgoals may be done several times, leading to redundant work and to infinite branches, thus preventing termination and completeness of proof procedures.

SLD resolution suffers from the same problem. Recognizing the similarity of subqueries in SLD trees, one could avoid some infinite branches. Furthermore, new occurrences of a subgoal could be resolved against existing answers of that subgoal, to cut down on computation. The above two ideas form the basis of the *memoing* techniques in classical Horn logic [MS81, Loz85, TS86, Die87, Vie87, Vie88, Vie89]. The latest advance in memoing techniques is *SLD-AL resolution*, proposed by Vieille as an extension of SLD resolution [Vie87, Vie88, Vie89].

SLD-AL resolution has been recognized as a powerful framework for query processing, and it is therefore worth investigating how to extend it to Horn–$\mathcal{T}_\mathcal{R}$. In the following, we extend $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ and SLD-AL resolution to *SLD-AL*$^{\mathcal{T}_\mathcal{R}}$ *resolution*, showing that memoing has good potential for establishing termination for a large class of transactions (by cutting off infinite branches), and enhancing performance (by avoiding recomputing similar goals).

SLD-AL Resolution

SLD-AL resolution retains the advantages offered by SLD resolution, and additionally constructs *finite* and complete search spaces (called *SLD-AL trees*), for finite and function free databases. It is therefore a powerful framework for query processing and optimization.

SLD-AL resolution rests on two principles. The *admissibility test* checks for similarity between goals, and cuts-off infinite branches in the finite and function-free case. *Non-admissible* subgoals are resolved against *previously derived answers* (referred to as *lemmas*), yielding new answers. These two mechanisms comprise the *AL-technique*, which works as follows: Let α' be a *variant* of an existing goal, α (i.e., α' and α differ at most in the names of their variables). If α is an ancestor of α', that is, α' is part of the evaluation of α, then further resolution of α' with input clauses leads to the construction of an infinite branch that repeats (indefinitely) variants of goal α. The admissibility test detects the similarity of these two goals and prevents resolution of α' with input clauses, by setting the *status* of α' to *non-admissible*. Instead, existing answers for α are used for resolving α'. Specifically, let η be a substitution, such that $\alpha\eta$ is a an answer (lemma) for α, that unifies with α'. Then $\alpha\eta$ can be thought of as a unit clause, $\leftarrow \alpha\eta$, and can thus be used for resolving α'. If no answers for α that unify with α' exist, then processing of goal α' is suspended, and processing of α is resumed, until some answers are produced or no further inferences are possible.

Note that, to avoid the infinite branch, one could follow a sequence of derivations other than the one which leads from α to α'. Reusing existing answers, however, is simpler and avoids redundant computation. In fact, to allow for more resolutions with lemmas and more gains in performance, one could use lemma resolution if α' is not a descendant of α, replace the *variant* notion with the *instance* notion, or do both. Consequently, there are four possible ways for defining a node as non-admissible.

The only restriction is that *local* selection functions must be used for SLD-AL resolution to be defined correctly. A local selection function selects one of the *most recently introduced* literals. Thus, if a node, N, is of the form $\leftarrow \alpha \;\&\; rest$, where α is the selected goal, then the descendants of N are of the form $\leftarrow (\beta \;\&\; rest)\theta$, where β contains only literals introduced directly or indirectly by α. In this way, processing is geared towards a proof for α. When a node $N' \equiv\leftarrow rest\eta$ is constructed, then the path in the tree from node N to N' (denoted N-N') is said to be a *proof segment* for α. Specifically, N-N' proves the lemma $\alpha\eta$. A local selection function is convenient for keeping track of proof segments, since it constructs them as compact sequences of nodes.

Definition 6.5 N-N' is called a *proof segment* for subgoal α (for N), iff N' is lower than N on the tree and N' is the first node on this branch that does not contain any descendant of α. If η is the current substitution at N', N-N' is said to *prove* the *lemma* $\alpha\eta$. N' is a *proof node* for $\alpha\eta$. $\Box$

Example 6.6 Consider again the program P_2 of Example 6.3

$$r(X,Y) \leftarrow e(X,Z) \;\&\; r(Z,Y).$$
$$r(X,Y) \leftarrow e(X,Y).$$

with the tuples $e(1,3)$, $e(3,1)$, $e(4,2)$ and $e(2,1)$ in the relation $EDGE$, and the query $? - r(X,1)$. Figure 6.2 depicts an (infinite) SLD-tree for this query. Its infinite branch is the result of the continuous resolution of goals of the form $r(X,1)$. Note also that the

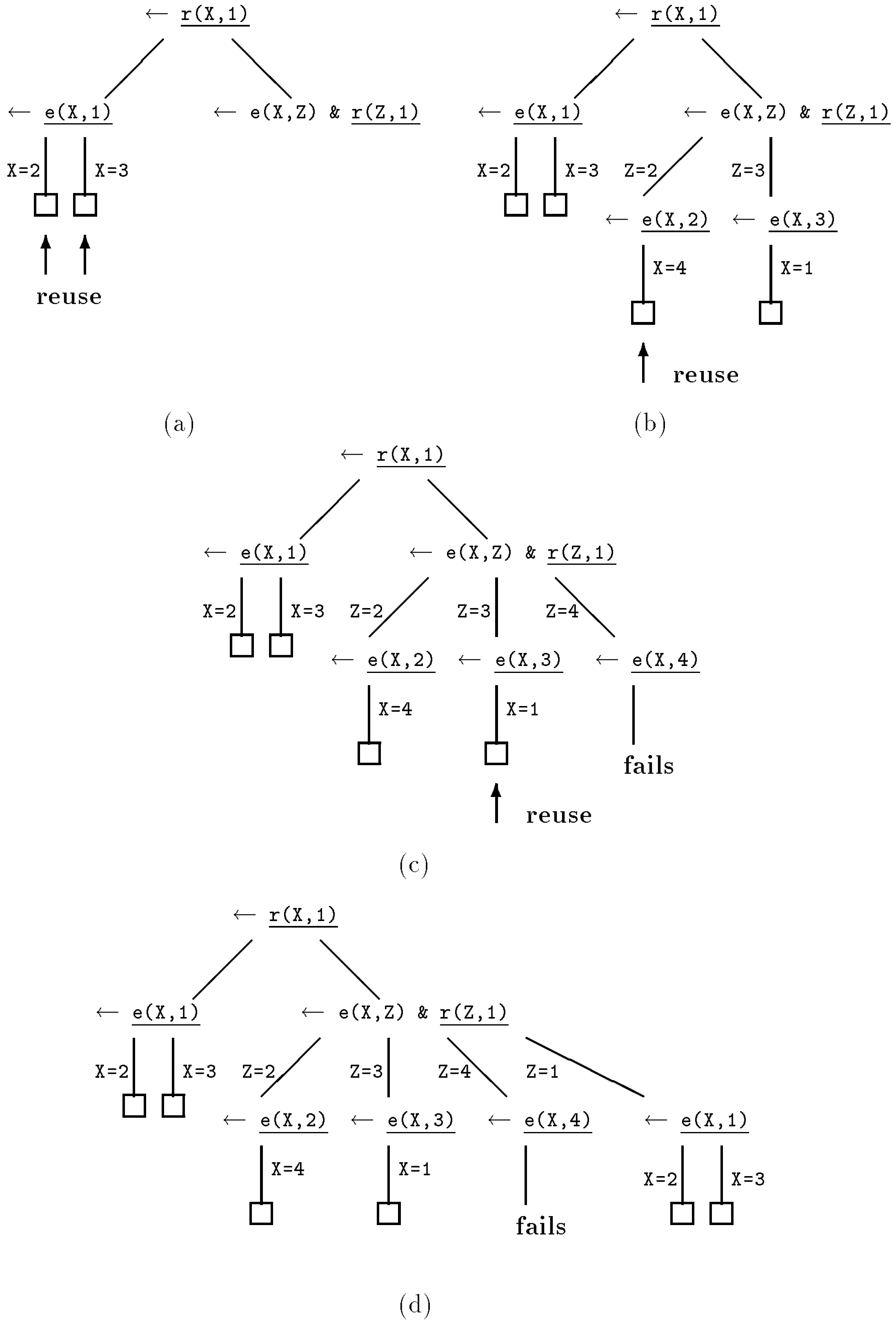

Figure 6.5: SLD-AL tree construction for Example 6.6.

data graph for *EDGE* is cyclic, thus causing all SLD trees for this query be infinite.

Figure 6.5 shows how SLD-AL resolution cuts off infinite branches. Resolving the query with the first rule, the node $\leftarrow e(X,Z)\ \&\ r(Z,1)$ is derived. If we choose $r(Z,1)$ as its selected atom, then this node becomes *non-admissible*, since $r(Z,1)$ is an instance of the existing goal $r(X,1)$. Thus, its processing is suspended, until some answers for $r(X,1)$ are produced by following other branches of the tree. Resolving the query with the second rule, we get the answers (*lemmas*) $r(2,1)$ and $r(3,1)$ (Figure 6.5.a). These lemmas unify with goal $r(Z,1)$. This means that they are also obtained if $r(Z,1)$ is resolved with the second rule. Instead of repeating the corresponding computation, lemmas $r(2,1)$ and $r(3,1)$ are reused to process further the suspended node. By applying them to resolve $r(Z,1)$, the new lemmas $r(4,1)$ and $r(1,1)$ are derived (Figure 6.5.b). By repeating this process, lemma $r(4,1)$ leads to failure (Figure 6.5.c), and lemma $r(1,1)$ does not produce any *new* lemmas (Figure 6.5.d). Thus, the construction and search of the SLD-AL tree is complete.

Note that resolution with lemmas resembles forward chaining. Thus, in a sense, SLD-AL resolution embeds bottom-up functionality to SLD resolution, and it is therefore considered a combined method. □

Example 6.7 (*Local selection functions are necessary)* Consider again the rules of Example 6.6, but assume that *EDGE* contains the tuples $e(1,2)$, $e(2,3)$ and $e(3,1)$. Consider now the transaction $? - ir(X,1)\ \&\ e(1,X)$. This has the correct answer $Y = 3$. Figure 6.6 shows what happens if we do *not* use a local selection function: Node (2) is non-admissible, since $r(Z,1)$ is an instance of $r(X,1)$. So its processing is suspended, waiting for $r(X,1)$ to provide lemmas. The selection of the literal $e(1,X)$ at node (3), however, leads to failure without yielding any answers for $ir(1,Y)$. Thus, the transactions fails, since there are no more inferences to make. Although the subtree rooted at node (3) does not produce any answers to the top goal, it is necessary for establishing a proof for $r(X,1)$. That could happen if a local selection function is used, that is, if $e(X,1)$ (introduced by $r(X,1)$), was the selected atom at that node. □

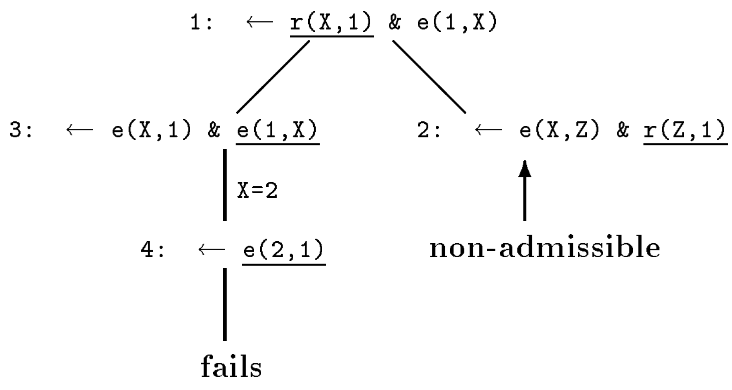

Figure 6.6: Local selection functions are necessary.

6.2.2 SLD-AL$^{\mathcal{T}_\mathcal{R}}$ Resolution: An Informal Presentation

SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution differs from SLD-AL resolution in the following two points. First, in Horn–$\mathcal{T}_\mathcal{R}$, the repetition of a goal alone is not an indication of the construction of an infinite branch, since the same transaction yields different results on different databases. Thus, the *database* component is taken into account for the admissibility test. Second, memoing and lemma resolution take place on *paths*, rather than database states, since answers are given on paths. Thus, we represent lemmas as pairs of the form (α, Δ), where α is a ground atom, and Δ is a database sequence.

Consider two nodes, N and N', of an SLD$^{\mathcal{T}_\mathcal{R}}$ tree evaluated on databases $\mathbf{D}$ and $\mathbf{D}'$, respectively, with selected atoms α and α'. Assume also that N was constructed before N', and that α' is an instance of α (i.e., $\alpha' = \alpha\theta$, for some substitution θ). Then, it might be possible that an answer for α, $(\eta, \mathbf{D}, \mathbf{D}_2, \ldots, \mathbf{D}_n)$, can be used for resolving α'. For instance, this is true when $\mathbf{D}$ and $\mathbf{D}'$ are equal, and α' unifies with $\alpha\eta$. When α' is a descendant and variant of α, this is crucial for avoiding the construction of an infinite branch that repeats variants of α on the database $\mathbf{D}$. In this case N' becomes *non-admissible*, and it is resolved only with answers for α. The derivation that results from the resolution with the above lemma is labeled with the unifying substitution, θ, and the database sequence $\mathbf{D}_2, \ldots, \mathbf{D}_n$.

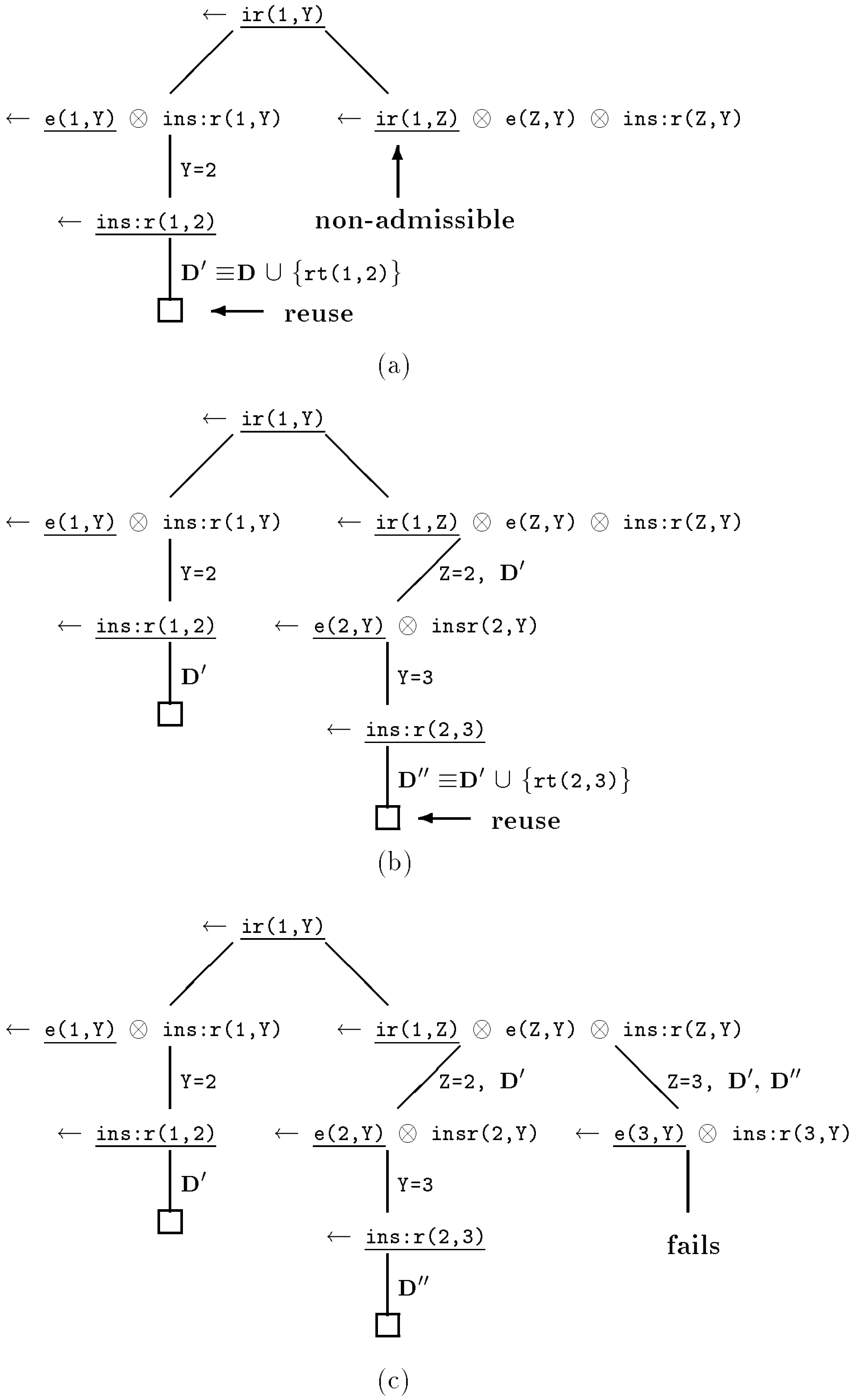

Figure 6.7: SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree construction for Example 6.8.

Example 6.8 Consider again the rules

$$(r1) \qquad ir(X,Y) \leftarrow ir(X,Z) \otimes e(Z,Y) \otimes ins{:}rt(Z,Y).$$

$$(r2) \qquad ir(X,Y) \leftarrow e(X,Y) \otimes ins{:}rt(X,Y).$$

where *EDGE* contains precisely the tuples $e(1,2)$ and $e(2,3)$, and the query $?-ir(1,Y)$. The SLD$^{\mathcal{T_R}}$ tree for this query has an infinite branch that repeats variants of the above goal, and it therefore prevents termination and (possibly) completeness.

Figure 6.7 shows how SLD-AL$^{\mathcal{T_R}}$ resolution solves the problem. Let $\mathbf{D}$ be the initial database. Then, as shown in Figure 6.7.a, the left branch of the constructed SLD-AL$^{\mathcal{T_R}}$ tree yields the lemma:

$$(ir(1,2),\ \mathbf{D}, \mathbf{D} \cup \{rt(1,2)\}).$$

The right branch leads to a node with selected atom $ir(1,Z)$, which is an instance of the root, $ir(1,Y)$. Thus, instead of resolving $ir(1,Z)$ with rule $r2$, $ir(1,Z)$ becomes non-admissible, and the above lemma is used for the resolution. This lemma can be thought of as the elementary transition

$$\langle \mathbf{D},\ \mathbf{D} \cup \{rt(1,2)\}\rangle ir(1,2).$$

The unifying substitution is $Z = 2$, the derived database is $\mathbf{D} \cup \{rt(1,2)\}$, and the derived node is

$$\leftarrow e(2,Y) \otimes ins{:}rt(2,Y).$$

As shown in Figure 6.7.b, this leads to a successful branch deriving a new lemma,

$$(ir(1,3),\ \mathbf{D}, \mathbf{D} \cup \{rt(1,2)\}, \mathbf{D} \cup \{rt(1,2), rt(2,3)\}).$$

This lemma can be thought of as the "transition"

$$\langle \mathbf{D},\ \mathbf{D} \cup \{rt(1,2)\},\ \mathbf{D} \cup \{rt(1,2), rt(2,3)\}\rangle ir(1,3).$$

In other words, resolution with this lemma takes place as shown above, except that it derives a database *sequence* rather than a database. The application of this new lemma does not return further answers, however, so the construction of the tree terminates (Figure 6.7.c) □

6.2.3 Admissibility Tests and Lemma Resolution

Let us now examine the circumstances under which memoing can be used. Our interest focuses on the case where new occurrences of a goal, α, are descendants of α (this is the case that the problem of infinite branch construction arise). In this case, in classical Horn logic, a non-admissible node has two properties. First, it is an indication of the presence of an infinite branch. Second, it produces all possible answers through lemma resolution. Thus, if there exist no lemmas that can be applied to a non-admissible node, then its answer is No. As a result, once an infinite branch is detected, it is cut off.

In contrast, in Horn–$\mathcal{T}_\mathcal{R}$ we might detect an infinite branch, without being able to cut it off. Detection alone is important, because we can then avoid infinite branches by following other branches (which lead to other path-answers). The non-existence of lemmas for a non-admissible node should not necessarily yield a No answer. Therefore, in SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution we distinguish between detecting an infinite branch and cutting it off. We suggest that two different admissibility tests should be used. The first extends the admissibility test in SLD-AL resolution. It detects infinite branches, and avoids them by resolving non-admissible nodes only with lemmas. The second is a generalization of the former. It merely detects a non-finite branch, and it marks nodes as *weakly non-admissible*. A weakly non-admissible node is *allowed* to be resolved with clauses, but preferably when all other alternatives are exhausted. In the following, we elaborate on the above discussion.

Non-Admissible Nodes

Example 6.8 presented a special case of a non-admissible node, since the databases for goals $ir(1, Z)$ and $ir(1, Z)$ were identical. Checking for database identity is trivial, since only the database identifiers need be compared. It is also naive, since loops in executions, which cause new occurrences of a transaction goal to appear on "similar" databases, remain unidentified. Database similarity can be defined in a number of ways. For instance, one could define it as database *equality*. For the purpose of infinite branch detection, this definition can be refined. This is because a transaction depends on only a small portion of the database. Assume that a transaction repeats itself on different databases, but the database portion that it depends on remains the same. Then it is possible that there is an infinite branch in the $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ tree. When database equality is used, the infinite branch is not recognized. Checking database equality, however, has two nice properties. First, lemma resolution is straightforward for equal databases. Second, it produces *all* possible answers for a non-admissible node. As that goal need not be resolved with clauses, the corresponding infinite branch is cut off.

Example 6.9 (*Detecting loops in execution*) Consider the program

$$(r1) \qquad a(X) \leftarrow b(X).$$
$$(r2) \qquad a(X) \leftarrow ins{:}c \otimes d(X).$$
$$(r3) \qquad d(X) \leftarrow del{:}c \otimes a(X).$$

$$\langle \mathbf{D}, \mathbf{D}' \rangle ins{:}c$$
$$\langle \mathbf{D}', \mathbf{D} \rangle del{:}c$$
$$\ldots$$

and suppose that the initial database is $\mathbf{D}$. The $\mathrm{SLD}^{\mathcal{T}_\mathcal{R}}$ tree for $\leftarrow a$ is shown in Figure 6.4. The right branch repeats indefinitely the initial goal. The goal at node (5) is the same as the initial one, and it is evaluated on the same database. Its database identifier, however, $\mathbf{D}''$, is not the same as $\mathbf{D}$, because the proof procedure uses a different descriptor every time an update occurs. Thus, unless the database at node (5) is compared with the one at node (1), the equality of databases and goals will pass unobserved. $\square$

Thus, we propose the following definition for non-admissible nodes. We would like to emphasize that the given conditions are not necessary, but *sufficient*, for finding nodes that can be resolved exclusively with lemmas.

Definition 6.6 (*Non-admissible node/goal*) Let the selected atom, α', at node N', be an *instance* of one of its *ancestors*, α, at node N, and let the databases at these nodes be *equal*. Then N' and α' are said to be *non-admissible*. α (N) becomes a *producer* and α' (N') becomes a *consumer* of answers provided by its producer. □

The Duplicate Elimination Problem

Note that the necessity for database comparisons gives rise to the *duplicate elimination* problem. For example, consider the transaction definition

$(r1)$ $\quad t \leftarrow ins{:}tuple \otimes t.$

$(r2)$ $\quad t \leftarrow query.$

Assume that the update *ins:tuple* adds a certain tuple to the database, regardless of the existence of that tuple in the database. Database equality might not be detected, due to tuple duplicates. Then an infinite branch is constructed by resolving continuously with rule $r1$. Thus, duplicates should be prevented, or database comparison should handle them.

Lemma Resolution for Non-Admissible Nodes

In SLD-AL$^{\mathcal{T_R}}$ resolution, nodes and databases are derived with respect to the node status (i.e., admissible or non-admissible). The definition for admissible nodes is the same as in SLD$^{\mathcal{T_R}}$ resolution. Non-admissible nodes, however, can be resolved with lemmas, *only*. Thus, we extend the definition for SLD$^{\mathcal{T_R}}$ resolution in Section 6.1 as follows:

- If N_i is *non-admissible*:

  Let $d_i \equiv (\alpha, \mathbf{D}_1, \ldots, \mathbf{D}_n)$ be an existing lemma such that $\mathbf{D}_1 \equiv \mathbf{D}_j$, and α, α_s are unifiable with mgu θ. Then

$$N_{i+1} \equiv \leftarrow (\ [\ \ldots \&\ \alpha_{s-1}\ \&\ \alpha_{s+1}\ \& \ldots\ ] \otimes rest\ )\theta,$$

  and N_{i+1} is evaluated on $\mathbf{D}_n$.

Weakly Non-Admissible Nodes

We would like to relax the database equality condition to give a test which is more successful in detecting non-finite branches. The key idea rests on the fact that a transaction depends on a fraction of the database, i.e., if this fraction remains the same, the transaction executes in the same way. The following example shows a case where it is possible to detect an infinite branch, although the admissibility test of Definition 6.6 fails. It is difficult, however, to come up with a test that detects similar cases. Nevertheless, it is an interesting problem that remains as a consideration for future work.

Example 6.10 Consider the following two programs.

$$\begin{aligned} P_1 :\ & ir(X,Y,I) \leftarrow e(X,Z) \otimes ins{:}rt(X,Z,I) \otimes ir(Z,Y,I+1).\\ & ir(X,Y,I) \leftarrow e(X,Y) \otimes ins{:}rt(X,Y,I).\\ & t(X,Y) \leftarrow ir(X,Y,0). \end{aligned}$$

$$\begin{aligned} P_2 :\ & ir(X,Y) \leftarrow e(X,Z) \otimes cnt(I) \otimes ins{:}rt(X,Z,I) \otimes incr{:}cnt \otimes ir(Z,Y).\\ & ir(X,Y) \leftarrow e(X,Y) \otimes cnt(I) \otimes ins{:}rt(X,Y,I).\\ & t(X,Y) \leftarrow set{:}cnt(0) \otimes ir(X,Y). \end{aligned}$$

The transaction $?-t(1,Y)$ has the same effect for both programs: It finds a path in the data graph for relation *EDGE* which starts from node 1, and inserts that path in the relation *ROUTE*. The third component of the tuples inserted in *ROUTE* is used for maintaining the order of edges in the chosen path. P_1 achieves this effect by using a counter which is incremented in the rule body. P_2 uses a counter in the database, stored a s a tuple, $cnt(i)$, for the same purpose. However, the increment operation is performed by the elementary update *incr*:*cnt*.

Given that *EDGE* contains precisely the tuples $e(1,2)$ and $e(2,1)$, both programs give rise to an infinite branch. Program P_1 produces an infinite sequence of the goals $\leftarrow ir(1,Y,2i)$ and $\leftarrow ir(2,Y,2i+1)$, for $i = 0,1,\ldots$, and program P_2 repeats the goals $\leftarrow ir(1,Y)$ and $\leftarrow ir(2,Y)$. Furthermore, note that a *new* database is derived every time *ins*:*rt* is executed, because of the insertion of invented (i.e., new) constants. The infinite branch created by P_1 is hard to detect, since it does not satisfy the requirement for goal similarity.

Non-finiteness in program P_2, however, is tractable. First, the non-finiteness is due to the repetition of a single goal, $ir(1,Y)$. Second, although the database changes constantly, the relation *EDGE* remains invariant, since the updates merely manipulate the tuple in the relation *CNT*. The subgoal $ir(1,Y)$ however, depends exclusively on the subgoal $e(X,1)$. This is because the subgoal $ir(Z,Y)$ receives its bindings for its first argument from the subgoal $e(X,Z)$. Thus, the repetition of $ir(1,Y)$ is due to the repetition of $e(X,1)$. Since *EDGE* remains the same, the same sequence of derivations that led to the repetition of $e(X,1)$ can be reproduced (unless a failure occurs in the mean time), and reproduce $e(X,1)$, and consequently $ir(1,Y)$. Since the latter remains the same throughout execution, the goal $ir(1,Y)$ is bound to be repeated indefinitely. $\square$

Lemma Resolution for Weakly Non-Admissible Nodes

There are two problems with weakly non-admissible nodes pertaining to lemma resolution. First, lemma resolution becomes complicated, since it involves database transformations. For example, let $\mathbf{D}$ be the database associated with a weakly non-admissible goal, α', and let $(\alpha,\ \mathbf{D}_1,\ldots,\mathbf{D}_n)$ be a resolving lemma. If $\mathbf{D}$ and $\mathbf{D}_1$ differ, then the sequence $\mathbf{D}_1, \ldots, \mathbf{D}_n$ must be modified to reflect execution when the lemma is not used. In particular, $\mathbf{D}_1$ must be replaced with $\mathbf{D}$. Such database transformations are, however, case dependent and in general undecidable.

Second, we do not always know whether a non-admissible node should be resolved exclusively with lemmas. Therefore, if no resolving lemmas exist, its answer is not necessarily *No*. It might be necessary to follow the infinite branch for a while, until the database reaches a state which has a successful derivation. We cannot determine, however, a safe upper bound on the length of the part of the infinite branch which must be constructed in order to derive the above state.

Example 6.11 Consider the transaction definition

$$(r1) \quad t \leftarrow \alpha \otimes t.$$

$$(r2) \quad t \leftarrow \beta.$$

The application of rule $r1$ consecutively and indefinitely yields either a failed or an infinite branch. Thus, the second occurrence of t in the branch is weakly non-admissible. Assume that α and β are independent, and that the initial database is $\mathbf{D}_0 = \mathbf{D}_0^\alpha \cup \mathbf{D}_0^\beta$, where $\mathbf{D}_0^\alpha$ is the part manipulated by α, and $\mathbf{D}_0^\beta$ is that manipulated by β. Then, an answer obtained by resolving goal t with rule $r2$ can be passed to goal t on any database that is derived directly or indirectly from the initial database by using rule $r1$, as follows: Let $\mathbf{D}_i^\alpha \cup \mathbf{D}_0^\beta$ be a database derived after i applications of rule $r1$, and let $(\eta, \mathbf{D}_0^\alpha \cup \mathbf{D}_0^\beta, \ldots, \mathbf{D}_0^\alpha \cup \mathbf{D}_k^\beta)$ be an answer obtained through $r2$. If β is executed on the database $\mathbf{D}_i^\alpha \cup \mathbf{D}_0^\beta$, taking exactly the same execution steps that led to the above lemma, then it will leave $\mathbf{D}_i^\alpha$ unchanged, and it will transform $\mathbf{D}_0^\beta$ according to the above lemma. Thus, another answer obtained by "reusing" this lemma is

$$(\eta,\ \mathbf{D}_0^\alpha \cup \mathbf{D}_0^\beta, \mathbf{D}_1^\alpha \cup \mathbf{D}_0^\beta, \ldots, \mathbf{D}_i^\alpha \cup \mathbf{D}_0^\beta, \mathbf{D}_i^\alpha \cup \mathbf{D}_1^\beta, \ldots, \mathbf{D}_i^\alpha \cup \mathbf{D}_k^\beta).$$

□

Example 6.12 Consider again the program of Example 6.11. Assume that β fails on $\mathbf{D}_0^\alpha \cup \mathbf{D}_0^\beta$. Then, since $\mathbf{D}_0^\beta$ remains unchanged by α, all databases produced are of the form $\mathbf{D}_i^\alpha \cup \mathbf{D}_0^\beta$. Thus, we can determine that the second occurrence of t is *non-admissible*, and suspend further processing, thus returning No. □

Example 6.13 Consider now the program P_2 of Example 6.10, together with the transaction definition

$$ird(X,Y,D) \leftarrow t(X,Y) \otimes rt(X,Y,D)$$

and the query $?-ird(1,Y,99)$. Given the data $e(1,2)$ and $e(2,1)$, this query is executed successfully. In order to succeed, however, the first 50 occurrences of $ir(1,Z)$ in the infinite branch are constructed, and then the second rule is applied once. □

6.2.4 SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees

For SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree construction we adopt the approach taken in [Vie89]. When constructing an SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree, we apply the AL-technique only to a subset of the predicates of the language, called the *r-predicates*. This class includes all recursive predicates. A goal (lemma) built on an r-predicate is called an *r-subgoal* (*r-lemma*). The construction of the tree involves the parallel generation of two sets, namely a collection of r-subgoals, $\mathbf{S}$, and a collection of r-lemmas, $\mathbf{L}$. The set $\mathbf{S}$ includes all the producers, and $\mathbf{L}$ consists of all the r-lemmas that have been proven so far. In the construction of SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees, we consider *admissible*, *non-admissible*, and *inactive* nodes, but for simplicity, we do not consider weakly non-admissible nodes. We require that *every* node belongs to one of the above three categories. Once a node is labeled as inactive, it is never processed again. We also refer to admissible and non-admissible nodes as *active*.

Definition 6.7 (*SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree construction)* Let $P = (\mathcal{B}, \mathbf{P}, \mathbf{D})$ be a definite program and N a definite goal. An SLD$^{\mathcal{T}_\mathcal{R}}$ tree for N is a tree with root N whose nodes are negative or empty clauses. Each arc is labeled with a database *sequence* and/or a substitution, and each node is associated with a database, namely the last database of the label of the first arc in the path from that node to the root. The construction of the tree and the aforementioned collections, **S** and **L**, is carried out as follows:

Initial step. N becomes the root of the tree, it is associated with the initial database, **D**, and it is taken to be admissible.

Current step. Choose an active node, N_i, with selected atom α_s, and *process* it as follows:

1. Apply a resolution which has not been performed yet (according to its status). If the resolution succeeds, add a child N_{i+1} to N_i resulting from the resolution. Let the selected atom in N_{i+1} be α'_s.

2. If N_{i+1} is a proof node for an r-lemma, λ, then add λ to **L** iff it is not an instance of a lemma already there.

3. Perform this step if α'_s is an r-subgoal. If α'_s is an instance of a subgoal α''_s in **S**, then mark α'_s as non-admissible and α''_s as a producer (if it is not already); otherwise, N_{i+1} is admissible and α'_s is added to **S**.

□

A Finiteness Property of SLD-AL$^{\mathcal{T_R}}$ Trees

The search space cannot be finite when the number of path-answers is not finite. Thus, SLD-AL$^{\mathcal{T_R}}$ trees are in general non-finite search spaces. They are not as large, however, as SLD$^{\mathcal{T_R}}$ trees, so they are preferable to SLD$^{\mathcal{T_R}}$ trees. Their advantage is that they cut off infinite branches in a number of cases. In the following, we specify a class of definite $\mathcal{T_R}$ programs for which SLD-AL$^{\mathcal{T_R}}$ trees do not have infinite branches. This property will appear to be crucial for the design of complete and effective proof procedures.

Theorem 6.4 (*Finiteness of SLD-AL$^{\mathcal{T_R}}$ tree branches)* If a $\mathcal{T_R}$-program is finite and function free, and the elementary updates add a finite number of constants to the database, then *all* SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees have *finite* branches.

Proof First note that the maximal size of **S** is bounded: Let b be the number of constants appearing in the initial database, the transaction base and the definitions of elementary updates. If c is an upper bound on the arity of EDB predicates, then we have at most b^c different tuples in each EDB relation. Thus, each EDB relation has at most 2^{b^c} instances. If we have k EDB relations, then we have at most $(2^{b^c})^k$ databases. Let a be an upper bound on the arity of r-predicates. Then, each goal can have at most a variables. Thus, each r-predicate has at most $(b+a)^a$ distinct (i.e., non-similar) goals. Hence, if we have m r-predicates, then we have at most $m(b+a)^a(2^{b^c})^k$ different *goals on databases*. This is, however, the maximal size of **S**.

Now, let the *depth* of a node be its number of ancestors, and let l be the maximal number of literals in the body of a clause. Then, the maximal depth is bounded by the maximal size of **S**, and the number of nodes for each depth, d, is bounded by l^d [Vie89]. The result follows then immediately from the boundness of **S**. $\square$

6.2.5 Search Strategies

Search strategies for SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution determine at each step of the SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree construction which node to resolve and which clause or lemma to resolve it with. A search strategy is *fair* if any potential resolution is selected after a finite number of steps [Vie89].[2] When a local selection function is used and the search strategy is fair, SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution is complete. Specifically, we *conjecture* the following theorem.

Theorem 6.5 (*Completeness of SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution*) Let P be a definite program, and N a definite goal. For any *local* selection function and any *fair* search strategy, if (θ, Δ) is a correct partial answer for N, there exists a computed answer, (σ, Δ), for N and a substitution, γ, such that $\theta = \sigma\gamma$. $\square$

[2] This definition of fairness is more appropriate than the definition of fairness of search strategies for SLD trees in [Llo87], since SLD trees are defined statically, while SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees are defined by specifying the way they are constructed.

A coarse characterization of search strategies for SLD-AL$^{\mathcal{T}_\mathcal{R}}$ resolution divides them into *2-phase* and *1-phase*. The 2-phase ones return a complete path-answer in two phases, as follows.

- *Phase 1.* Follow a successful path in the search space, and return the corresponding answer, (θ, Δ). This should be a depth-first search.
- *Phase 2.* Make the chosen partial path-answer complete, by following (from the root) all branches that yield the same database sequence, Δ.

1-phase strategies look for a complete answer by following all branches with the same execution path in parallel.

- *Step 1* Starting from the root, follow all branches that lead to the same database, say $\mathbf{D}_1$.
- *Step i* If a path-answer has not been found yet, then follow all branches that lead to the same database, $\mathbf{D}_{i+1}$.

In the process of constructing branches, those branches that result in failure are simply ignored. Only when all branches for the database sequence under construction fail, a different database sequence is followed by starting from the root again.

Unfortunately, not all search strategies are complete or effective. The effectiveness of a search strategy is determined mainly by the size of the constructed tree, the volume of database changes, and the demand for database reconstruction. Achieving completeness is hard even if all tree branches are finite. This is because the search strategy might be trapped in the computation of an infinite number of answers for a subgoal, thus failing to return all answers for the top query.

Example 6.14 Consider again the rules

$$ir(X,Y) \leftarrow ir(X,Z) \otimes e(Z,Y) \otimes ins{:}rt(Z,Y).$$
$$ir(X,Y) \leftarrow e(X,Y) \otimes ins{:}rt(X,Y).$$

and assume that *EDGE* contains in the initial database, $\mathbf{D}$, the tuples $e(1,2)$, $e(2,3)$ and $e(3,1)$. Let $\mathbf{D}' \equiv \mathbf{D} \cup \{rt(1,2)\}$, $\mathbf{D}'' \equiv \mathbf{D}' \cup \{rt(2,3)\}$, and $\mathbf{D}''' \equiv \mathbf{D}'' \cup \{rt(3,1)\}$. Then, the query $? - ir(1,Y)$ has the following infinite number of possible answers:

$$(\{2\},\ \mathbf{D},\mathbf{D}'),$$
$$(\{3\},\ \mathbf{D},\mathbf{D}',\mathbf{D}''),$$
$$(\{1\},\ \mathbf{D},\mathbf{D}',\mathbf{D}'',\mathbf{D}'''),$$
$$(\{2\},\ \mathbf{D},\mathbf{D}',\mathbf{D}'',\mathbf{D}''',\mathbf{D}'''),$$
$$\ldots$$

This is because the elementary update succeeds, even though it does not change $\mathbf{D}'''$. Thus, the corresponding SLD-AL$^{\mathcal{T}_\mathcal{R}}$ tree (which is constructed exactly as described in Example 6.8 and Figure 6.7 is infinite. Its infiniteness results from the generation of an *infinite number* of *finite* branches, each leading to a different path-answer. Specifically, $ir(1,Z)$ resolved with a lemma,

$$(ir(1,i),\ \mathbf{D},\mathbf{D}',\mathbf{D}'',\mathbf{D}''',\mathbf{D}''',\ldots) \quad (k \text{ occurrences of } \mathbf{D}'''),$$

generates a new lemma,

$$(ir(1, i \ mod\ 3\ +1),\ \mathbf{D},\mathbf{D}',\mathbf{D}'',\mathbf{D}''',\mathbf{D}''',\ldots) \quad (k+1 \text{ occurrences of } \mathbf{D}''').$$

Consider now the query $? - ir(1,Y) \otimes e(Y,1)$. It has an infinite number of answers, namely the answers (Θ,Δ) with substitution $\Theta = \{3\}$. If however, the search strategy gets stuck in the generation of answers for the subgoal $ir(1,Y)$, then no answer will ever be returned to the top query. $\square$

A Multistage Depth-first Strategy

A *Multistage depth-first strategy* is proposed in [TS86] to handle function symbols. We define a multistage depth-first strategy for SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution below, modifying slightly the definition given in [TS86]. We show that the multistage depth-first strategy guarantees completeness when all branches are *finite*.

Definition 6.8 (*Multistage depth-first strategy)* First assume the following ordering on the nodes of the SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ tree: The children of a node are ordered by the order of creation; a node, N, is to *the left* of a node, N', if N precedes N' in the left-to-right post-order traversal of the tree.

The search process consists of several *stages*, each of which has a number of *steps.* In each step the *leftmost active* node is chosen (thus the mode of search is depth-first), for processing as described in Definition 6.7. If it is non-admissible, however, then the resolution is limited to those lemmas *which are generated in the previous or earlier stages*. When there are no nodes to be processed, the next stage begins. The entire process terminates when no more new facts are produced. □

Theorem 6.6 (*Completeness of the multistage depth-first strategy)* If all branches of the SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ tree are *finite*, then the multistage depth-first strategy is *complete*.

Proof According to Theorem 6.5, it suffices to prove that the multistage depth-first strategy is *fair*. Thus, it suffices to show that each stage applies all resolutions which are possible *so far*. We prove this result by induction on the number of stages. Specifically, we prove that for each stage, s, the input rule base (i.e., the set of program clauses and lemmas) and the constructed tree are finite. This implies that each stage applies *all* possible resolutions in a finite number of steps.

Basis. The input rule base for the first stage is finite. Thus, since each node can be resolved with a finite number of clauses, the tree constructed by applying all possible resolutions available so far is finite in breadth. By our assumption, it is also finite in depth. Hence, the constructed tree is finite.

Induction step. Assume the results hold for stage s. We prove that they also hold for stage $s+1$. The input rule base is the input rule base of stage s plus the lemmas generated in that stage. Each producer in stage s had a finite number of consumers, and the number of producers was also finite (since the constructed tree is finite, by the induction hypothesis). Thus, each lemma in stage s yielded a finite number of lemmas when applied to the corresponding consumers. Since the number of input lemmas was finite, the number of produced lemmas was also finite. Thus, the input rule base for stage $s+1$ is finite. The output tree of stage s is the input tree for stage $s+1$, and, by the induction hypothesis, it is finite. Thus, given that the rule base is finite, the constructed tree is finite in breadth. It is thus finite, because each branch has a finite length. $\square$

Example 6.15 Consider again Example 6.14. Figure 6.8 shows the first two stages for executing $?-ir(1,Y)\otimes e(Y,1)$. The node numbering indicates the order of creation. Note that an answer, $(\{3\},\ \mathbf{D},\mathbf{D}',\mathbf{D}'')$, for the top goal is generated by resolving node (8). Node (8) is a proof node for the lemma $(ir(1,3),\ \mathbf{D},\mathbf{D}',\mathbf{D}'')$. A strategy which is not multistage could decide to reuse this lemma, thus preventing the generation of the above answer for the top goal. $\square$

A Strategy for Eliminating Lemma Redundancy

Consider again Example 6.14, with the query $?-ir(1,Y)$. Assume that the leftmost branch is followed, and the partial path-answer $(2,\ \mathbf{D},\mathbf{D}')$ is obtained. Since we have decided that $\mathbf{D},\mathbf{D}'$ is the execution path, we must prevent the search strategy from constructing branches on other paths. When the lemma $(ir(1,2),\ \mathbf{D},\mathbf{D}')$ is applied to $ir(1,Z)$, it produces the answer $(3,\ \mathbf{D},\mathbf{D}',\mathbf{D}'')$, which has a path of greater length. This is true for all lemmas. Thus, it is obvious that $(\{2\},\ \mathbf{D},\mathbf{D}')$ is a *complete* answer.

Let us now elaborate on the above argument, specifying a strategy for optimizing lemma resolution. Assume that the (chosen) execution path is $E=\mathbf{D}_0,\mathbf{D}_1,\ldots,\mathbf{D}_n$. Let two goals, α and α' be associated with the databases $\mathbf{D}_i$ and $\mathbf{D}_{i+k}$, respectively. Suppose

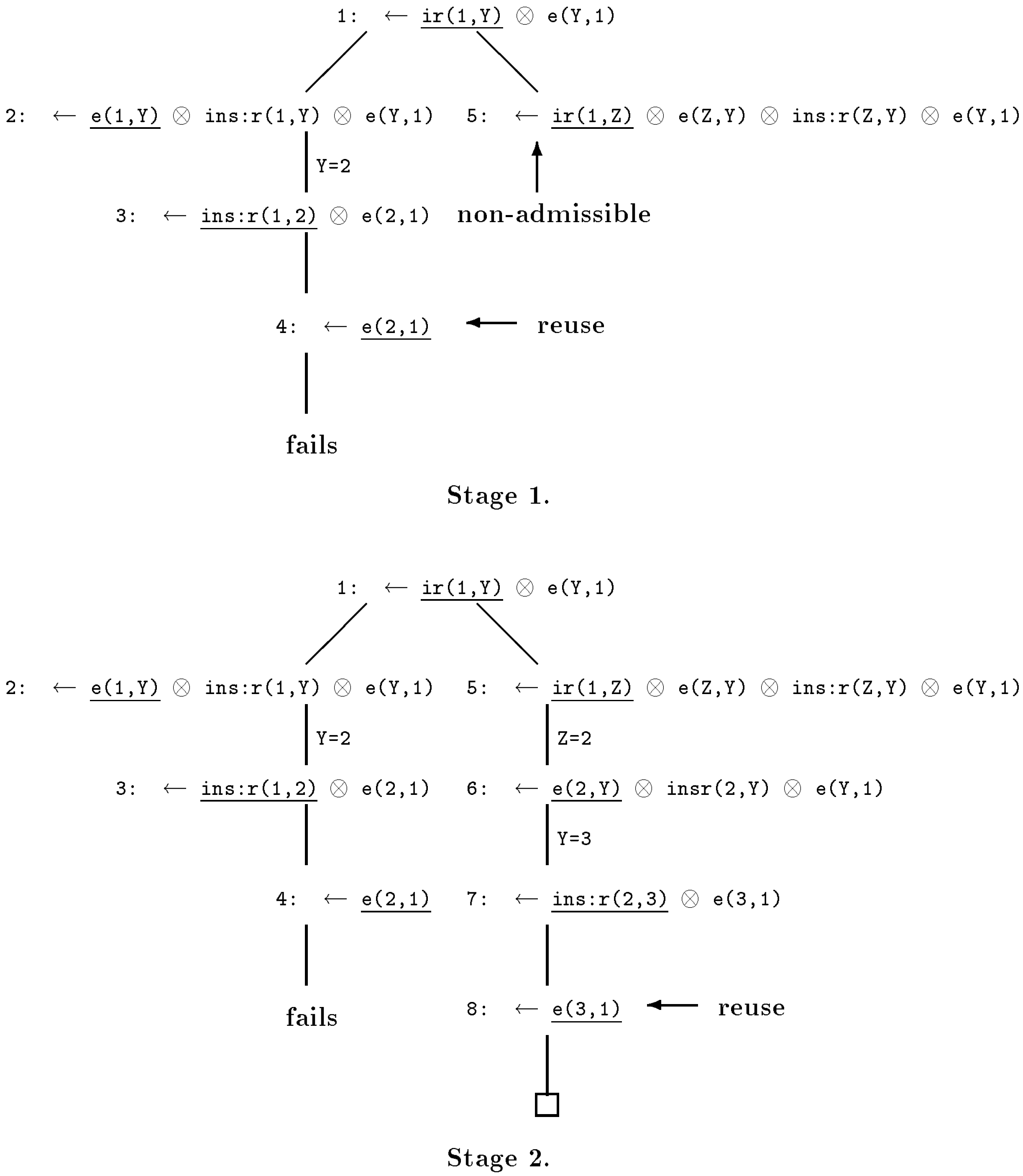

Figure 6.8: Using a multistage depth-first strategy.

that α' is non-admissible, having α as producer (thus, $\mathbf{D}_i \equiv \mathbf{D}_{i+k}$). Let $(\alpha\theta, \Delta)$ be a lemma for α. If $(\alpha\theta, \Delta)$ were applied to α', then the resulting child node of α' would be on the path $(\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_{i+k}) \circ \Delta$. In order that this child node be considered for processing, Δ must be a subsequence of $\mathbf{D}_{i+k}, \mathbf{D}_{i+k+1}, \ldots, \mathbf{D}_n$. Assume that this is not true, i.e., that $(\alpha\theta, \Delta)$ does not contribute an answer to the top query on the path E, when it is applied to α'. Then, a lemma, $(\alpha\eta, \Delta')$, generated by applying $(\alpha\theta, \Delta)$ to α', does not contribute such an answer, since $(\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_{i+k}) \circ \Delta$ would be a subsequence of Δ', implying that Δ' would not be a subsequence of E. Consequently, $(\alpha\theta, \Delta)$ should *not* be considered for resolving α' if Δ is not a subsequence of $\mathbf{D}_{i+k}, \mathbf{D}_{i+k+1}, \ldots, \mathbf{D}_n$. This means that when we have a finite number of lemmas for α on each path, then the number of children of α' is finite.

It is easy to generalize the above discussion when α has m consumers, $\alpha^{(1)}, \ldots, \alpha^{(m)}$. Then, a lemma, λ, which is not useful when applied to one consumer, $\alpha^{(i)}$, might be useful when applied to another, $\alpha^{(j)}$. Furthermore, it is possible that the lemma derived by applying λ to $\alpha^{(i)}$ is also useful when it is applied to $\alpha^{(j)}$. However, a lemma can be safely *not* considered for further resolutions when it is useful for *none* of the consumers $\alpha^{(1)}, \ldots, \alpha^{(m)}$. Therefore, $\alpha^{(1)}, \ldots, \alpha^{(m)}$ are guaranteed to have a finite number of children when the number of lemmas on each path is finite. Then, given that all admissible nodes have a finite number of children (since the number of rules for each predicate is finite), the constructed search space *is finite in breadth*.

Theorem 6.7 When all branches of an SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ tree are *finite*, there is a refutation procedure which returns a *complete* path-answer. Furthermore, in the finite and function-free case it also *terminates*.

Proof By Theorem 6.6, the multistage depth-first strategy guarantees to find a partial path-answer, (θ, Δ), if one exists, when all branches are finite. Thus, the refutation procedure can choose Δ as the execution path. Then it suffices to combine the multistage depth-first strategy with the lemma redundancy elimination strategy. □

6.3 Conclusion

We addressed the problem of the evaluation of definite $\mathcal{T}_{\mathcal{R}}$ programs in a top-down fashion. We developed two sound and complete evaluation frameworks, namely SLD$^{\mathcal{T}_{\mathcal{R}}}$ and SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution, by extending SLD and SLD-AL resolution, respectively.

In contrast to bottom-up evaluation, proof procedures that are based on these frameworks are able to focus on *one* path-answer. They also lead to better space utilization, since they can make use of efficient, stack based memory management, in contrast to bottom-up evaluation that uses space proportional to the number of derived facts. They are therefore particularly suited to non-deterministic transactions.

The most interesting results concern SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution: We showed that SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution leads to complete and terminating proof procedures, for a certain class of programs. Furthermore, SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution has great optimizations potential. In [Vie89] Vieille develops techniques for eliminating redundancy in SLD-AL trees, and shows that they are powerful enough to account for the best known recursive query evaluation methods. These techniques can be extended to SLD-AL$^{\mathcal{T}_{\mathcal{R}}}$ resolution.

We draw the conclusion that top-down evaluation beats bottom-up for non-deterministic transactions. Their comparison for deterministic transactions is an open problem.

Chapter 7

Conclusions and Directions for Further Research

In the preceding chapters we cast light on the semantics of Horn–$\mathcal{T}_\mathcal{R}$ programs, complementing the analysis of Bonner and Kifer [BK93], and investigated the evaluation of these programs. We showed that existing evaluation methods for classical Horn logic can be extended in a natural way to Horn–$\mathcal{T}_\mathcal{R}$, providing frameworks for both bottom-up and top-down evaluation. We provided solutions for a number of problems related to the path semantics of the logic, the execution of updates, non-deterministic and serial execution, and discussed open research issues.

Section 7.1 sums up the contributions of this thesis. Section 7.2 considers a number of research issues as directions for future work.

7.1 Results

The problems dealt with in this thesis and the solutions presented are summarized as follows:

- *The specification of the intended meaning of Horn–$\mathcal{T}_\mathcal{R}$ programs.* We approached the problem from two points of view. First, from a model-theoretic point of view, we defined the intended meaning of a Horn–$\mathcal{T}_\mathcal{R}$ program, P, as the intersection of all Herbrand models of P. We called this model the "least Herbrand model" of P, and proved it to consist of precisely those facts that are entailed by the formulae in P. Second, we provided a fixpoint semantics, which casts more light on the meaning of Horn–$\mathcal{T}_\mathcal{R}$ programs, and proved it to be equivalent to the model-theoretic semantics.

These results enhance the theoretical foundation of [BK93] for transaction execution.

- *The investigation of bottom-up evaluation methods for Horn–$\mathcal{T}_\mathcal{R}$ programs.* The objectives were the achievement of good performance and the exploitation of existing technology for bottom-up evaluation of classical Horn programs. The primary problems we faced were the representation of facts on paths, execution on paths, and execution of updates. We proposed a solution which is based on the fixpoint semantics of Horn–$\mathcal{T}_\mathcal{R}$.

  We first set up a framework for bottom-up evaluation. We used relations to represent facts on paths, and we showed that path composition, which accounts for execution on paths, reduces to the natural join operation. This approach enabled us to use standard relational algebra operations for evaluating rules. Second, we showed how to handle execution of elementary updates. In effect, we showed that bottom-up evaluation methods for classical Horn programs could be used with moderate modifications for evaluating Horn–$\mathcal{T}_\mathcal{R}$ programs.

  As discussed, performance of pure bottom-up evaluation is not satisfactory in the presence of updates (especially non-deterministic updates), because of the explosion of the number of databases, facts, and procedure calls to elementary updates. Thus, our objective was to enable bottom-up evaluation to focus on both relevant data and relevant databases. In order to focus on relevant databases, we first devised a rule rewriting technique that extended ordinary goals with database arguments. Then, being able to refer explicitly to the database input and output of transactions, we showed that classical rule rewriting techniques fulfilled the objective.

Despite the incorporated top-down control, bottom-up evaluation cannot focus on one path answer, since it computes the totality of the least fixpoint. It is therefore bound to perform poorly for non-deterministic execution.

- *The investigation of top-down evaluation for Horn–$\mathcal{T}_\mathcal{R}$ programs.* We approached the problem by extending two frameworks for top-down evaluation of classical Horn programs, namely SLD and SLD-AL resolution. In particular, we extended SLD resolution to $\text{SLD}^{\mathcal{T}_\mathcal{R}}$ resolution and proved it to be sound and complete. Unlike bottom-up evaluation, $\text{SLD}^{\mathcal{T}_\mathcal{R}}$ proof procedures were shown to focus on one path answer. Like their SLD counterparts, however, they were not necessarily complete. We thus extended $\text{SLD}^{\mathcal{T}_\mathcal{R}}$ resolution to $\text{SLD-AL}^{\mathcal{T}_\mathcal{R}}$ resolution. The extension included infinite branch detection and reuse of previously computed answers. In effect, $\text{SLD-AL}^{\mathcal{T}_\mathcal{R}}$ resolution resulted in complete proof procedures for certain classes of transactions and enhanced performance. $\text{SLD}^{\mathcal{T}_\mathcal{R}}$ resolution has great optimizations potential, and it is likely the best choice for developing an interpreter for Horn–$\mathcal{T}_\mathcal{R}$.

7.2 Directions for Future Research

There are many research problems that still remain open after this work, concerning especially optimization issues. We consider the following as directions for future research:

- Data replication. Since an update affects only a portion of the database, we expect evaluation methods to maintain database differentials, rather than complete databases. The problem is challenging, because the objective is to combine optimal storage with the ability to perform quick database comparisons, rollbacks, and queries on several databases.

- Elimination of redundancy in SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees. It is of interest to extend existing techniques for reducing the search space, such as local and global optimization for SLD-AL trees [Vie88, Vie89], to SLD-AL$^{\mathcal{T}_\mathcal{R}}$ trees. Heuristics should also be devised for pruning those branches that do not yield answers on the chosen execution path, to optimize non-deterministic execution.

- Rule changes and non-determinism in bottom-up evaluation. As discussed in Chapter 5, rule changes cannot be handled in the proposed bottom-up evaluation framework. It is thus interesting to modify bottom-up evaluation, so that it does.

  Non-determinism is the weak point of bottom-up evaluation for Horn–$\mathcal{T}_\mathcal{R}$. This is because bottom-up evaluation computes the totality of the least fixpoint, thus generating all possible path-answers for non-deterministic transactions. It is challenging to develop techniques for enabling bottom-up evaluation to focus on one path answer.

Bibliography

[AV88] S. Abiteboul and V. Vianu. Procedural and Declarative Database Update Languages. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 240–250, March 1988.

[AvE82] K. R. Apt and M. H. van Emden. Contributions to the theory of Logic Programming. *Journal of ACM*, 29(3):841–862, July 1982.

[BK93] Anthony J. Bonner and Michael Kifer. Transaction Logic Programming. Technical Report CSRI-270, Computer Systems Research Institute, University of Toronto, May 1993.

[BLV93] P. Bayer, A. Lefebvre, and L. Vieille. Architecture and Design of the EKS Deductive Database System. *Submitted to the special issue of the VLDB Journal on Prototypes of Deductive Database Systems*, 1993.

[BMSU87] F. Bancilhon, D. Maier, U. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. 5th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–15, 1987.

[BR87] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 269–283, 1987.

[BR88] F. Bancilhon and R. Ramakrishnan. Perfomance Evaluation of Data Intensive Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 439–511. Morgan Kaufman, 1988.

[Bra91] I. Bratko. *Prolog programming for artificial intelligence.* Addison-Wesley, Wokingham, England, 2nd edition, 1991.

[CKRP73] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un Systeme de Communication Homme-Machine en Francais, 1973. Groupe de Recherche en Intelligence Artificielle, Université d' Aix-Marseille.

[Die87] S. W. Dietrich. Extension tables: memo relations in logic programming. In *Proc. 1987 Symp. on Logic Programming*, pages 264–272. IEEE Computer Society, 1987.

[Elk90] Charles Elkan. Independence of Logic Queries and Updates. In *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 154–160, April 1990.

[End72] H. B. Enderton. *A mathematical introduction to logic.* Academic Press, New York, 1972.

[GMN84] H. Gallaire., J. Minker, and J. M. Nicholas. Logic and databases: a deductive approach. *ACM Computing Surveys*, 16(2), June 1984.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *ACM SIGMOD Int. Conf. On Management of Data*, pages 157–166, 1993.

[GN87] M. Genesereth and N. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann, 1987.

[KM90] H. Katsuno and A. O. Mendelzon. On the Difference between Updating a Knowledge Base and Revising it. Technical Report KRR-TR-90-6, Department of Computer Science, University of Toronto, Toronto, 1990.

[Kow74] R. A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*, pages 569–574, 1974.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[Loz85] E. L. Lozinskii. Evaluating queries in a deductive database by generating. In *Proc. 9th IJCAI*, pages 173–177, 1985.

[MS81] D. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings 7th IJCAI*, pages 368–374, 1981.

[NK88] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. 7th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 251–262, March 1988.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Kowledge Bases.* Computer Science Press, 1989.

[Ram88] R. Ramakrishnan. Magic Templates: A Spelling Approach to Logic Programs. In *Proc. 5th International Conference and Symp. on Logic Programming*, pages 140–159, 1988.

[Rei92] R. Reiter. On Specifying Database Updates. Technical Report KRR-TR-92-3, Department of Computer Science, University of Toronto, Toronto, 1992.

[RFV83] J. D. Ullman R. Fagin and M. Y. Vardi. On the Semantics of Updates in Databases. In *Proc. of the 2nd ACM Symp. on the Principles of Database Systems*, pages 16–33, April 1983.

[RLK86] J. Rohmer, R. Lescouer, and J. M. Kerisit. The Alexander Method – A Technique for the Processing of Recursive Axioms in Deductive Databases. *New Generation Computing*, 4(3):273–285, 1986.

[Rob65] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1965.

[Sek89] H. Seki. On the Power of Alexander Templates. In *Proc. 8th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 150–159, March 1989.

[Shm87] O. Shmueli. Decidability and Expressiveness Aspects of Logic Queries. In *Proc. 6th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 237–249, 1987.

[TS86] H. Tamaki and T. Sato. OLD Resolution with Tabulation. In *Proc. 3rd Int. Conf. on Logic Programming*, pages 84–98, 1986.

[Ull89a] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, Rockville, Maryland, 1989.

[Ull89b] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 2. Computer Science Press, Rockville, Maryland, 1989.

[vEK76] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of ACM*, 23(4):733–742, October 1976.

[vG86] A. van Gelder. A Message Passing Framework for Logical Query Evaluation. *ACM SIGMOD Int. Conf. On Management of Data*, pages 155–165, 1986.

[Vie87] L. Vieille. Database-Complete Proof Procedures Based on SLD Resolution. In *Proc. 4th Int. Conf. on Logic Programming*, pages 74–103, May 1987.

[Vie88] L. Vieille. From QSQ Towards QoSaQ: Global Optimization of Recursive Queries. In *Proc. 2nd Int. Conf. on Expert Database Systems*, pages 421–434, 1988.

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

[Win88] M. Winslett. A Model-Based Approach to Updating Databases with Incomplete Information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.