The University of New South Wales

School of Computer Science and Engineering

Inductive Logic Programming

Phuong Nguyen (2200798)

Bachelor of Science (Computer Science)

November 2001

Supervisors: Prof. Arun Sharma, Dr. Eric Martin
Assessor: Dr. John Shepherd

Abstract

This thesis studies a new theory where we introduce a generalized notion of logical consequence in a new framework. This generalized notion of logical consequence captures the notions of "logical consequence", which have been present in a number of existing frameworks, such as Logic Programmings, ILP, nonmonotonic reasoning. Many results have been found regarding the properties of the generalized notion of logical consequence. These include a number of basic results, as well as some results about the relationship between the generalized notion of logical consequence and the existing notion appearing in nonmonotonic reasoning, in particular, the closed world assumption.

In our framework, we investigate the structure of the set of generalized logical consequences of a theory. Here the set of generalized logical consequences of a theory is structured in a hierarchy over the theory. A level in the hierarchy is obtained from the lower ones either by a deductive step or an inductive step. In particular, formulas on the second level correspond to the deductive consequences of the theory, while under some assumptions, formulas on the fourth level correspond to the inductive and deductive consequences of the theory. With some appropriate settings of the parameters of the framework, we obtain the working context of Prolog. Prolog, with its deductive inference ability, can be considered as providing a proof procedure for the first nonground level of the hierarchies. It has been used in a number of ILP systems, which aim at providing some sort of inductive inference mechanism. While relying on the deductive power of Prolog, these systems loose the declarativeness of Prolog. In this thesis, a system, *RichProlog*, has been built, which is capable of handling both deduction and induction. *RichProlog* therefore can be regarded as a proof procedure for the first four levels in the hierarchies. It will be seen how *RichProlog* maintains the declarative nature of Prolog, while providing both deductive and inductive inference abilities. Compared to other ILP system, this is a big advantage of *RichProlog*. The system is then used to solve the problem of learning nonerasing pattern languages from positive data only.

Acknowledgements

Firstly I would like to express my gratitude to Professor Arun Sharma, Doctor John Shepherd and the School of Computer Science and Engineering for providing funds for this honour's year. Without their help, this thesis would not be possible. I would like to thank Professor Arun Sharma for suggesting this topic, and Doctor Eric Martin for dedicating many hours enthusiastically supervising me on the topic.

I would like also to thank Professor Claude Sammut for his initial guidelines in studying the *iProlog* system. Conversations with David Eyers and Peter Gammie at CSE also helped me greatly in understanding the codes. Finally, Peter Gammie helped to ease the task of setting up the latex files for the report.

Contents

Chapter 1

Introduction

In classical logic, the notion of logical consequence is coupled with deduction. This is due to the soundness and completeness of the deduction system. Thanks to the compactness property, the deductive consequences are both certain and justifiable. However, this also shows that classical logic provides no justification for any kind of induction, and in particular, not a full foundation to Inductive Logic Programming. Inductive Logic Programming is concerned with inductive inference of logic programs from examples and background knowledge. The notion of induction, as it appears in Inductive Logic Programming ([MR94]), is based on the claim that induction is the reverse of deduction. Unlike deductive consequences, inductive consequences are neither certain nor justifiable. This is because induction makes claims on knowledge that go beyond the observed facts. Note that this notion of induction is also present in a number of frameworks such as learning theory ([OSW86]) or nonmonotonic reasoning ([Luk90]).

In this thesis, a general notion of logical consequence will be proposed. It encompasses both notions of deductive consequence and inductive consequence. While deduction is associated with compactness, induction is seen to be related to a weak form of compactness. This notion of logical consequence is obtained in a general logical framework, where a number of other frameworks are seen as special instances. For example, by specializing this logical framework, one can obtain classical logic, learning paradigms and their formalization of inductive inference, as well as a number of other frameworks such as various settings of nonmonotonic reasoning. This general framework therefore provides a foundation for a unified logic of deduction and induction. As a result, the generalized notion of logical consequence will capture the classical notion of logical consequence, as well as other kinds of "logical consequence" which are present in, for example, the concept to be learned in a learning paradigm, or the consequences in some settings of nonmonotonic reasoning.

The generality of the logical framework is achieved by formalizing and abstracting a number of concepts that have been present (usually implicitly) in the existing frameworks. Let consider, for example, the field of nonmonotonic reasoning. Here induction can be conceived as follows. For a theory T, we would like to be able to infer some sort of "logical consequences" of T, which do not follow deductively from T. In particular, the naive closed world assumption ([Luk90]) achieves this by adding into the theory T literals whose negations do not follow from T. The effect of this process is that the enlarged theory, if it is consistent at all, will have exactly one minimal Herbrand model. The intended consequences of T are then exactly the formulas which are true in this minimal Herbrand model. It can be seen that this process is repeated in other settings of the closed world assumption. In all cases, the results are similar. We always arrive at the situation where the closure of a theory has some (maybe one) unique Herbrand models, which are minimal in some sense. As another example, in the field of ILP, most settings require the background theory and hypotheses to be definite logic programs. The result is that the theories in these settings have unique minimal Herbrand models. As can be seen, these efforts spent in constructing the closures of the theories, or requiring the definity of the clauses, all aim at restricting the set of models of a theory to some intended models. Analysing this fact gives us the following. Firstly, the desired models are of some restricted kind i.e., the Herbrand models. Secondly, the desired models must also satisfy some kind of minimality condition. It is therefore a natural process to formalize these concepts, and to integrate them into a general, unified framework. In the next parts, we will briefly introduce the components of this general framework.

1.1 A generalized notion of logical consequence

The notion of logical consequence is generalized by abstracting a class of possible worlds and formalizing a notion of minimality. These concepts have been present, either implicitly or explicitly, in a number of frameworks. Let us first examine some natural examples for the class of possible worlds. In classical logic, it is implicitly taken to be the class of all structures. In the field of ILP one mainly works with the set of all Herbrand structures. The definite setting, as mentioned above, restricts the background theory and the hypotheses to be definite, thus the theories in this case have unique minimal Herbrand models. Other settings of ILP, such as the nonmonotonic setting, also work with minimal Herbrand models of the theories. As discussed above, in the field of nonmonotonic reasoning, various forms of the closed world assumption implicitly impose the models of the theories to be taken from the set of Herbrand structures. In the field of AI, possible worlds can be the Henkin structures, where individuals are interpreted by closed terms.

Next consider the formalization of the notion of minimality. It is defined on the basis of a set $\mathcal{O}$ of possible observations. Minimality is then conceived as characterizing a model of a theory in terms of its extension in $\mathcal{O}$. This is a formalization of what may happen implicitly in other frameworks. For example, in some interpretations of the closed world assumption, the Herbrand models of a theory are compared with respect to their extension in $\mathcal{O}$, where $\mathcal{O}$ can be the Herbrand base (naive closed world assumption), or some sets of basic sentences (e.g. careful closed world assumption). Another example is in Logic Programming, where the focus is on the (unique) minimal Herbrand models that satisfy a given set of clauses. These minimal Herbrand models minimize the set of atomic sentences that are true in it, hence the set of all possible observations can be taken as the set of all atomic sentences.

The notion of logical consequence is then redefined with respect to the class $\mathcal{W}$ of possible worlds and the notion of minimality, or essentially the set $\mathcal{O}$ of possible observations. The generalized logical consequences of a set T of formulas are the formulas which are true in all models of T taken from the class of possible worlds, and which are minimal with respect to the defined notion of minimality. It will not be surprising now that if we take appropriate values for $\mathcal{W}$ and $\mathcal{O}$, we will obtain the existing frameworks. This is shown in the results presented in the appendix. It can be seen that there is a whole family of generalized logical consequences parameterized by $\mathcal{W}$ and $\mathcal{O}$. As a result, any proof system will be specific to some choice of $\mathcal{W}$ and $\mathcal{O}$, and for other settings of $\mathcal{W}$ and $\mathcal{O}$, no such inference system is known. The structure of the set of generalized logical consequences of a theory is complex and has not been thoroughfully understood. In the next section we will introduce the notion of a general logical framework, which will help to study the structure of the set of generalized logical consequences of a theory.

1.2 The general framework

The classical notion of logical consequence has been well investigated. The deduction systems, such as the sequent calculus ([EFT89]) are both sound and complete. These give a complete view on the structure of the set of logical consequences of a theory in the classical framework. There is no such system for the generalized notion of logical consequence. Various attempts have been made to study the properties of many other notions of logical consequence. Some theoretical results have been implemented in various systems. For example, a kind of closed world assumption is reflected in Logic Programmings by implementing negation as failure. By making these assumptions, some forms of logical consequences are realized by deductive inference. Another example is in the field of ILP, where rules of inductive inference and operators ([MR94]) have been introduced.

Our approach is to examine the set of generalized logical consequences of a theory with reference to other theories. This leads to the formalization of a set of possible theories $\mathcal{T}$. As it turns out, focusing on a particular set of possible theories $\mathcal{T}$ provides comprehensive picture of the sets of generalized logical consequences of the theories in $\mathcal{T}$. In particular, theories in

$\mathcal{T}$ structure the set of generalized logical consequences of each of its members. Generalized logical consequences of a theory will then be seen to form a hierarchical structure. In this hierarchy, the height of the level where a formula resides reflects the difficulty of deriving it from the theory. More precisely, the higher the level where a formula φ appears, the more difficult it is to derive φ from the underlying theory. Obviously, different choices of $\mathcal{T}$ provide different classifications of the generalized logical consequences of its members i.e., they give different hierarchies. Not surprisingly, some formulas may appear in high levels in some hierarchies, and low levels in some others. This reflects the fact that the formulas might be easier to be derived from some theories than others. In addition, some hierarchies can be incomplete in the sense that they do not contain all generalized logical consequences of the theories. This can be seen as reflecting the "power" of the set of possible theories. An example of the possible theories is found in the working context of Prolog where they can be the sets of definite Horn clauses. As another example, they can be the theories whose naive closures are consistent (in the naive closed world assumption).

In more detail, the levels in the hierarchy of generalized logical consequences of a theory correspond to an alternation of deductive and inductive inferences. The first level consists of the theory itself. The next level contains the deductive consequences of the theory. Then the third level is obtained by an inductive step, and the forth level is obtained by a deductive step, etc. This can also be seen as an alternation of a compactness and a weak compactness property. This approach where we introduce the notion of a set of possible theories also provides the theoretical foundation for a system that will be introduced in the next section. This system is able to perform both induction and deduction. It has been built, and will be described in this thesis. We will now introduce a new notion of consistency, which is paired with the generalized notion of logical consequence.

The first step in the formalization of a set of possible theories requires the generalized notion of consistency, which is a result of introducing the generalized notion of logical consequence. Naturally, a theory is consistent if it has at least one minimal model taken from the class of possible worlds. It can be seen that this notion of consistency is much stronger than the classical one. Some sets of formulas can be consistent in the classical setting, but not according to the new notion. The possible theories are required to be consistent with respect to this stronger notion.

So far we have presented the elements of the general framework. It is parameterized firstly with the class of possible worlds. Classical logic implicitly works with the class of all structures, while other frameworks may consider only the set of Herbrand structures or the set of Henkin structures. The framework also consists of the set of possible observations, which provides the basis for the notion of minimality. In some cases, the set of possible observations might be the set of all atomic sentences, or a subset of it. These notions of a class of possible worlds and the set of possible observations define the generalized notion of logical consequence as well as the new notion of consistency. We also focus on some consistent sets of formulas, determining the last parameter of the general framework: the set of possible theories. Within this framework, we are able to investigate the structure of the set of generalized logical consequences of a theory. More precisely, they form a hierarchy over the theory. Different settings of the framework provide different hierarchies. Some may be powerful enough to build complete hierarchy of all generalized logical consequences of its theories, while others are not. In a hierarchy, the second level corresponds to the deductive consequences, while the forth level corresponds to both deductive and inductive consequences of the theories. Within the setting of Logic Programming, the second level is handled by Prolog. We will now turn our attention to a system that can handle the first four levels in an extension of the setting of Prolog.

1.3 *RichProlog*

As mentioned above, the hierarchy over a theory T is such that a higher level is obtained from the lower ones either by a deductive inference or an inductive inference, corresponding to alternating compactness and weak compactness properties. The first level is the theory T itself. The next level (namely level Σ_1) is obtained from T by a deductive step, and thus contains the deductive consequences of T. The third level (i.e. level Π_1) is obtained from level Σ_1 by an inductive step. Then the next level (i.e. level Σ_2) is obtained from level Π_1 by a deductive step, etc. Intuitively, the higher the level a formula φ appears, the more difficult it is to derive φ from T. Formulas in level Σ_1 include the classical deductive consequences of T, while under some assumption on the parameters of the logical framework, formulas in level Σ_2 can be shown to correspond to the concepts that are learnable in the limit in a learning paradigm.

The computation in Prolog is designed to be a side effect of a proof procedure that determines whether a formula φ is a deductive consequence of a theory T. Prolog therefore can be considered as implementing a Σ_1 proof procedure. In this thesis, Prolog will be extended to a system *RichProlog* whose computation will be a side effect of a proof procedure that determines whether a formula φ is a deductive or an inductive consequence of a theory T. Basically this system will be able to handle queries occurring in Σ_2, Π_1 as well as Σ_1 levels. Inputs to the system will be sets of clauses (disjunctions of basic formulas), which can be regarded as the background knowledge, with optionally the evidence (basic formulas). The queries are expected to be formulas in the levels Σ_2, Π_1 and Σ_1, or essentially formulas in Σ_2, Π_1 and Σ_1 prenex form ([EFT89]). In the following paragraphs we will mainly focus on queries in level Σ_2.

Basically, Prolog handles queries of the form $\exists\overline{x}\varphi$ (where $\exists\overline{x}$ is a denotation for $(x_1, \ldots, x_n)$, and φ is a quantifier-free formula). It does this by looking for a witness for the existence of $\overline{x}$. Formulas in the level Σ_2 are of the form $\exists\overline{x}\forall\overline{y}\varphi$, or equivalent to some formulas of this form, where φ does not contain quantifiers. (There is a closed relationship between being in level Σ_n (resp. Π_m) and being in the Σ_n (resp. Π_m) prenex form, and this is one of the reasons why the levels in the hierarchy are so named.) *RichProlog* will handle these queries by looking for a witness for the existence of $\overline{x}$. Informally, the witness can be said to satisfy all instances of $\overline{y}$. This is not a trivial task. While searching for the witness of a Σ_1 query can be carried out using resolution, such a search is impossible for Σ_2 queries. The search for witness of a Σ_2 query relies on the basis of an enumeration of closed terms. Essentially, we assume that an enumeration of closed terms is available. We will find (or "generate") the possible witnesses that satisfy a finite subset of the instances of $\overline{y}$. These possible witnesses are then subject of a validation process, during which the "right" witnesses are selected. The inductive nature of the process is present when moving from possible witnesses (that satisfy a finite subset of instances of $\overline{y}$) to the right witnesses (that satisfy all instances of $\overline{y}$).

The finite subset of instances of $\overline{y}$ used in the inductive step is defined on the basis of a measure of the closed terms in the enumeration. Basically, this subset has to have a measure that exceeds some threshold θ. We will use I to denote this subset. Members of I will be used to generate possible witnesses. To generate a possible witness from a closed instance of $\overline{y}$, we can "plug" it into the formula $\exists\overline{x}\forall\overline{y}\varphi$ (i.e. replace the variables in $\overline{y}$ by the corresponding closed terms[1]) to obtain a Σ_1 formula, and then carrying out the resolution process. Note that by plugging a closed instance of $\overline{y}$ into $\exists\overline{x}\forall\overline{y}\varphi$ we obtain a Σ_1 formula, and we obtain different formulas when we plug different instances of $\overline{y}$ to φ. (If we depict the parse tree for these formulas, then they differ by having different subtrees whose roots are at the nodes which contain subinstances of $\overline{y}$.) Generating a possible witness is then done similarly as when trying to find a witness for the Σ_1 query. To generate possible witnesses which satisfy all members of I, we can plug them in φ in turns. The unifications of all possible witnesses generated by each member of I are the possible witnesses generated by I.

Prolog uses resolution with backtracking to find witness for Σ_1 queries. A similar procedure needs to be carried out in generating possible witnesses for the Σ_2 query $\exists\overline{x}\forall\overline{y}\varphi$ by each instance of $\overline{y}$ in the finite set I. Therefore generating possible witnesses from I can be seen as consisting of a number of applications of these processes. In other words, we say that the Σ_2 search consists of a number of stages, each of them being equivalent to a Σ_1 search, where we find the witnesses for a finite set of Σ_1 formulas. We can also say that the Σ_2 search procedure is obtained from the Σ_1 search procedure of Prolog, in such a way that when the witnesses are found, instead of stopping or printing them out, we move to the next stage. Once again, the possible witnesses generated by a set I of instances of $\overline{y}$ are the unifications

[1]Formally, this means that we obtain $\exists\overline{x}\varphi[\overline{y}/\overline{\beta}]$, which is the result of substituting the variables $y_1, \ldots, y_m$ in $\overline{y}$ by the terms $\beta_1, \ldots, \beta_m$ in β, the closed instance of $\overline{y}$ in the context

of the possible witnesses generated by each member of I. Thus in moving from one stage to the next, we can start with the formula φ where the possible witness found from the previous stage is plugged into φ. The possible witnesses found at the last stage are then naturally the possible witnesses generated by the whole set.

In order to select the right witnesses for the Σ_2 queries, we need to validate the possible witnesses. To say that $\overline{\alpha}$ is a witness for $\overline{x}$ in the query $\exists\overline{x}\forall\overline{y}\varphi$ is to say that $\forall\overline{y}\varphi[\overline{x}/\overline{\alpha}]$ is true. Thus to validate a possible witness $\overline{\alpha}$ is to check that the Π_1 formula $\forall\overline{y}\varphi[\overline{x}/\overline{\alpha}]$ is true. We can do this by refutation i.e., we can check that $\neg\forall\overline{y}\varphi[\overline{x}/\overline{\alpha}]$ is not true. Now $\neg\forall\overline{y}\varphi[\overline{x}/\overline{\alpha}]$ is equivalent to $\exists\overline{y}\neg\varphi[\overline{x}/\overline{\alpha}]$, and the refutation can be done by checking that the Σ_1 query $\exists\overline{y}\neg\varphi[\overline{x}/\overline{\alpha}]$ does not have any witness for $\overline{y}$. At this point, we are involved with the notion of true negation. This notion of true negation is different from negation as failure, which is usually implemented in Prolog systems. While the implementation of negation as failure in Prolog makes it possible to handle the queries of form $\neg\exists\overline{x}\psi$, or equivalently, queries of form $\forall\overline{x}\neg\psi$, Prolog is unable to handle queries of form $\exists\overline{x}\neg\psi$. Therefore the first difference between *RichProlog* and a Prolog system is *RichProlog*'s ability to handle true negation. Note that the notion of true negation is necessary, for it appears either in the process of generating possible witnesses, or in the validation process. We will see (in chapter 4) how *RichProlog* distinguishes between positive and negative literals, and then we will see how it applies resolution to handle formulas of both signs uniformly.

We have seen so far that each possible witness results in a Π_1 formula. To validate a possible witness we need to negate the corresponding Π_1 formula to obtain a Σ_1 formula, which can be handled by finding witness for it. Handling these Σ_1 formulas can be done using resolution (a modification of the process carried out by Prolog). It can be seen that the Σ_1 searches that are performed in order to generate witnesses for a Σ_2 query as discussed above use the same logic program. The logic program used here is actually derived from the input clauses and the formulas φ, where $\exists\overline{x}\forall\overline{y}\varphi$ is the query being handled. On the other hand, the logic program used in the Σ_1 search as performed in the validation process is derived from the input clauses and $\neg\varphi$. We will discuss how to construct these programs when we present the algorithms for a Σ_2 proof procedure in chapter 3.

As will be seen, true negation raises a number of issues that have to be addressed. For example, it implies that n different rules can be associated with a clause with n literals. For instance, the clause $\neg p \vee q$ has two rules associated with it, namely $q \leftarrow p$ and $\neg p \leftarrow \neg q$. When deriving the logic program from a formula and a set of clauses, we can potentially use all rules that are logically equivalent to the clauses (in reality, it is often the case that not all rules defined from a given clause are generated). Dealing with true negation raises another problem i.e., the logic programs may contain inconsistent rules, in that case the result derived from the program may not be reliable. Essentially this is the case when the program contains literals of the form φ and $\neg\psi$ where φ and ψ can unify. In technical terms, this is referred to as "the program contains potential conflicts".

We will address the issues mentioned so far when we describe *RichProlog*. Implementation of *RichProlog* is an extension of the implementation of *iProlog*, which has been developed at UNSW ([SS83b, SS83a]). True negation is achieved by modifying the data structure representing terms. We also need to modify the unification procedure to accommodate the negative formulas. The parsing procedure also needs to be modified to allow for the negation symbol, as well as queries of different prenex form. Derivation of different programs for a Σ_2 query $\exists\overline{x}\forall\overline{y}\varphi$ is done by selecting the rules associated with the input clauses. Checking for potential conflicts is done in this phase. Most importantly, the Σ_2 search is performed by modifying the existing Σ_1 search procedure (which is the prove procedure, as it is called). This is done as follows. At each stage, when a witness is found, instead of printing it, we advance to the next stage by plugging the next instance of $\overline{y}$ to φ. Within one stage, the backtracking is performed as it was in *iProlog*. However, we also need to backtrack between stages. This basically means that in the case when the current possible witness generated by an instance of $\overline{y}$ is refuted at a later stage, we need to try for another possible witness.

1.4 Layout of the report

The following chapters are organized as follows. In chapter 2, the general logical framework will be discussed in more detail. We will formally introduce the parameters of the paradigm. Then some of the basic results, together with their proofs will be presented. These are meant to give more insight into this new field of study. Investigation of the theory in the first phase of the thesis results in a number of discoveries. These have been submitted to the "Eighth International Conference on Principles of Knowledge Representation and Reasoning" (KR2002). They are presented in the appendix. To fully present the theory is beyond the scope of this thesis. One is therefore referred to the appendix, [MSS] and [MSS01].

Chapter 3 will describe the algorithms for a Σ_2 proof procedure. Being able to handle induction as well as deduction, it has some unique features. These include the ability to handle true negation and handle queries in level Π_1. They will be discussed in this chapter. An example will be shown to illustrate the arguments. This chapter also addresses issues in backtracking as it is used to simulate the nondeterminism of the algorithms. Finally, it will address some practical considerations in implementing the *RichProlog* system.

Chapter 4 will describe in detail the system *RichProlog*, which has been built in this thesis. This chapter begins with *iProlog*, a Prolog system which has been used as a starting point for the implementation of *RichProlog*. We then give an example to illustrate an execution of a Σ_2 query in *RichProlog*. It is followed by the description of the implementation of *RichProlog*. We will also discuss some possible improvements that can be made for the system. These include optimizations that can be applied to an ordinary Prolog system in general, and to *iProlog* in particular, as well as some exploitations of *RichProlog*'s unique features, such as inherent parallelism. We close the chapter with an application of *RichProlog* to a learning problem. This is the problem of learning nonerasing pattern languages from positive data only.

We then conclude the report with some remarks on the theory as well as the implementation of the system. These will identify issues that need to be addressed in future work.

Chapter 2

An introduction to the background theory

This chapter presents the theory on which this thesis is based. It involves a generalized notion of logical consequence. As a particular case, this notion of logical consequence coincides with the classical notion of logical consequence. In general, the set of (generalized) logical consequences of a theory has a complex hierarchical structure, which resembles the Borel hierarchy ([Sri98]). The first level of this hierarchy is the theory itself, while a higher level is obtained from the lower ones by an alternation of "deductive" and "inductive" steps.

This generalization is the result of parameterizing the logic with a class of possible worlds and a set of possible observations. Classical logic is the case where every structure is under consideration, and every sentence can be an observation. Logic Programming corresponds to the case where one is interested in the set of all Herbrand structures and the set of all atomic sentences. Instead of developing a separate, complete theory for each set of possible worlds and each set of possible observations, they are considered as the parameters in the framework under investigation. Other frameworks can then be cast into this framework by appropriate assignment of the parameters.

The layout of this chapter is as follows. First the basic concepts will be formalized. This is followed by some basic results, which aim at giving insight into the theory. Finally, a brief discussion about the hierarchy of the logical consequences of a theory will be presented.

2.1 Notation

A *vocabulary* is a countable set of function symbols (possibly including constants) and predicate symbols. It may or may not contain equality. A vocabulary is usually denoted by S. The set of all first-order formulas whose symbols are taken from S (S*-formulas*, or just *formulas*) is denoted by $\mathfrak{L}$. *Sentences*, or *closed formulas*, are formulas which do not contain variables. Subsets of $\mathfrak{L}$ are called *theories*.

The Σ_n and Π_n *prenex form* formulas are defined inductively as follows. Σ_0 and Π_0 formulas are quantifier-free members of $\mathfrak{L}$. Let $n > 0$, and $\varphi \in \mathfrak{L}$. φ is said to be in Σ_n (resp. Π_n) prenex form if one of the following holds:

- φ is in Σ_{n-1} or Π_{n-1} prenex form.
- φ is of form $\exists x\psi$ (resp. $\forall x\psi$) for some $\psi \in \mathfrak{L}$ which is in Σ_n (resp. Π_n) prenex form.

It can be shown that any member of $\mathfrak{L}$ is equivalent to a member of $\mathfrak{L}$which is in either Σ_n or Π_n prenex form (for a proof see [Ham78]).

Given a vocabulary S, the notion of S-structure and models can be found in [EFT89]. In this thesis, "structure" will often be used instead of "S-structure", given that the intended meaning is clear from the context. Following the convention,

$$\mathfrak{M} \models \varphi$$

will be used to denote that $\mathfrak{M}$ is a model of φ, or in other words, φ is true in $\mathfrak{M}$.

Given a set $\Gamma \subseteq \mathfrak{L}$, for different structures sometimes we are interested in members of Γ which are true in the structures. Thus, let $\mathfrak{M}$ be a structure, the Γ-diagram of $\mathfrak{M}$, denoted $D_\Gamma(\mathfrak{M})$, is the set of all members of Γ that are true in $\mathfrak{M}$:

$$D_\Gamma(\mathfrak{M}) = \{\varphi \mid \varphi \in \Gamma, \mathfrak{M} \models \varphi\}$$

2.1.1 Possible worlds

The generalized notion of logical consequence is obtained firstly by allowing the class of structures to vary. This introduces the concept of a *class of possible worlds*, which is a class of structures under consideration. From now on, $\mathcal{W}$ will be used to denote the class of possible worlds.

Let T be a theory. A member of $\mathcal{W}$ that is a model of T is called a *model of T in $\mathcal{W}$*. The class of all models of T in $\mathcal{W}$ is written as $\mathrm{Mod}_{\mathcal{W}}(T)$. For a $\varphi \in \mathfrak{L}$, we will use $\mathrm{Mod}_{\mathcal{W}}(\varphi)$ rather than $\mathrm{Mod}_{\mathcal{W}}(\{\varphi\})$. The notions of consistency and logical consequences are revised as follows.

Definition 2.1. *Let T be a theory. It is said to be* $\mathcal{W}$-consistent *if $Mod_{\mathcal{W}}(T) \neq \emptyset$, and* $\mathcal{W}$-inconsistent *otherwise.*

Definition 2.2. *Let T be a theory and $\varphi \in \mathcal{L}$. φ is said to be a* logical consequence of T in $\mathcal{W}$, *written $T \models_{\mathcal{W}} \varphi$, if and only if every member of $Mod_{\mathcal{W}}(T)$ is a model of φ.*

The set of all logical consequences of a theory T is denoted by $\mathrm{Cn}_{\mathcal{W}}(T)$. The following property is immediate from the definitions above. One should compare it with similar results in the classical setting. Basically it underlines the strong link between the notion of models and logical consequences: two theories have the same sets of models in $\mathcal{W}$ if and only if they have the same sets of logical consequences in $\mathcal{W}$.

Property 2.3. *For all theories T, T', the following are equivalent:*

- $Cn_{\mathcal{W}}(T) = Cn_{\mathcal{W}}(T')$.
- $Mod_{\mathcal{W}}(T) = Mod_{\mathcal{W}}(T')$.

If one lets $\mathcal{W}$ be the class of all structures then the generalized notions of consistency and logical consequence in $\mathcal{W}$ coincide with those in the classical setting. In general, $\mathcal{W}$-consistency is stronger than consistency. There are also nontrivial cases where the two notions coincide. [MSS] shows examples of both statements.

Another parameterization requires the concept of possible observation, which will be introduced next.

2.1.2 Possible observations

Many frameworks are not concerned with all models of a set of formulas. For example Logic Programming usually works with the minimal Herbrand models of a given set of clauses ([Doe94]). In these frameworks, the intended models must satisfy the minimality criteria. The notion of minimality is often relative to a set of closed formulas. To express this explicitly, let a set of closed formulas of $\mathcal{L}$ be denoted by $\mathcal{O}$. $\mathcal{O}$ is then called the set of *possible observations*. Possible choices for $\mathcal{O}$ include the set of all atomic sentences, the set of all basic sentences, or subsets of these.

The concept of minimality is formalized in the following definition. The intended models of a theory are the ones whose $\mathcal{O}$-diagrams are minimal compared to other models of that theory.

Definition 2.4. *Let T be a theory and $\mathfrak{M}$ be a structure. We say that $\mathfrak{M}$ is an* $\mathcal{O}$-minimal model of T in $\mathcal{W}$ *if and only if $\mathfrak{M} \in Mod_{\mathcal{W}}(T)$ and for all $\mathfrak{N} \in Mod_{\mathcal{W}}(T)$,*

$$D_{\mathcal{O}}(\mathfrak{N}) \not\subset D_{\mathcal{O}}(\mathfrak{M})$$

The following property follows immediately from the notation. It provides a characterization of the $\mathcal{O}$-minimal models in $\mathcal{W}$ of the $\mathcal{O}$-diagrams:

Property 2.5. *Let theory T be an $\mathcal{O}$-diagram and $\mathfrak{M} \in \mathcal{W}$. Then $\mathfrak{M}$ is an $\mathcal{O}$-minimal model of T in $\mathcal{W}$ if and only if $D_{\mathcal{O}}(\mathfrak{M}) = T$.*

For $T \subseteq \mathcal{L}$, the class of all structures that are $\mathcal{O}$-minimal models of T in $\mathcal{W}$ is denoted by $\mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T)$.

We now arrive at yet another stronger notion of consistency:

Definition 2.6. *Let $T \subseteq \mathcal{L}$. It is said to be* $\mathcal{O}$-minimally $\mathcal{W}$-consistent *if $Mod^{\mathcal{O}}_{\mathcal{W}}(T) \neq \emptyset$, and* $\mathcal{O}$-minimally $\mathcal{W}$-inconsistent *otherwise.*

There are cases where $\mathcal{O}$-minimal $\mathcal{W}$-consistency does not coincide with $\mathcal{W}$-consistency. Examples can be found in the appendix. The corresponding notion of logical consequence is as follows.

Definition 2.7. *Let $T \subseteq \mathcal{L}$ and $\varphi \in \mathcal{L}$. φ is said to be an* $\mathcal{O}$-minimal logical consequence of T in $\mathcal{W}$, *denoted $T \models^{\mathcal{O}}_{\mathcal{W}} \varphi$, if and only if every member of $Mod^{\mathcal{O}}_{\mathcal{W}}(T)$ is a model of φ.*

The set of all $\varphi \in \mathcal{L}$ such that $T \models^{\mathcal{O}}_{\mathcal{W}} \varphi$ is denoted by $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$.

2.1.3 Possible theories

With the introduction of $\mathcal{O}$-minimal $\mathcal{W}$-consistency, theories which are not $\mathcal{O}$-minimally $\mathcal{W}$-consistent are not of interest. This introduces the notion of *possible theories*. A possible theory is a set of closed members of $\mathcal{L}$ which are $\mathcal{O}$-minimally $\mathcal{W}$-consistent. A nonempty set of possible theories is denoted by $\mathcal{T}$. Letting the set of theories of concern vary gives another dimension of freedom to the framework. By specifying the set of possible theories, one focuses in only theories which have some desired properties. For example, in the case of logic programs the theories are the mix of possible observations with background knowledge. The complete picture of the framework is given in the definition in the next part.

2.1.4 Paradigms

All the parameters introduced so far are integrated into the following definition.

Definition 2.8. *A paradigm is a quadruple consisting of:*

1. *A vocabulary S;*
2. *A nonempty class $\mathcal{W}$ of S-structures;*
3. *A set $\mathcal{O}$ of closed members of $\mathcal{L}$;*

4. A nonempty set of sets of closed members of $\mathcal{L}$ that are $\mathcal{O}$-minimally $\mathcal{W}$-consistent.

The paradigm $(S, \mathcal{W}, \mathcal{O}, \mathcal{T})$ as defined above is denoted by $\mathcal{P}$.

In varying the parameters in $\mathcal{P}$ one obtains a number of settings that have been studied. For example, with a fixed vocabulary S, first-order logic corresponds to the case where one lets $\mathcal{W}$ equal to the class of all structures, $\mathcal{O} = \emptyset$ and $\mathcal{T}$ equal to the set of all consistent sets of closed members of $\mathcal{L}$.

Given these definitions and notations, the next part will present some basic results.

2.2 Some basic results

This section aims at providing some insight into the theory. It will present and discuss some basic results, mainly concerned with the notion of $\mathcal{O}$-minimality. One could find in [MSS] the relationship between various notions of model, consistency and logical consequence that have been defined above.

When the set of observations is closed under negation it has a unique property. In this case, each model in $\mathcal{W}$ of a theory is an $\mathcal{O}$-minimal model of that theory in $\mathcal{W}$. This is reflected in the following property.

Proposition 2.9. *Suppose that $\mathcal{O}$ is closed under negation. For every theory T of $\mathcal{L}$, $Mod^{\mathcal{O}}_{\mathcal{W}}(T) = Mod_{\mathcal{W}}(T)$.*

Proof. Let T be a theory. By definition, $\mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T) \subseteq \mathrm{Mod}_{\mathcal{W}}(T)$. For the other direction, we need to show that each model of T in $\mathcal{W}$ is also one of T's $\mathcal{O}$-minimal models in $\mathcal{W}$. Let $\mathfrak{M} \in \mathrm{Mod}_{\mathcal{W}}(T)$. We need to show that $\mathfrak{M} \in \mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T)$.

Suppose by the way of contradiction that there exists $\mathfrak{N} \in \mathrm{Mod}_{\mathcal{W}}(T)$ such that $D_{\mathcal{O}}(\mathfrak{N}) \subset D_{\mathcal{O}}(\mathfrak{M})$. Let φ be arbitrary in $D_{\mathcal{O}}(\mathfrak{M}) \setminus D_{\mathcal{O}}(\mathfrak{N})$. Then $\varphi \in \mathcal{O}$ and $\varphi \notin D_{\mathcal{O}}(\mathfrak{N})$, and therefore $\mathfrak{N} \models \neg\varphi$. Since $\mathcal{O}$ is closed under negation $\neg\varphi \in \mathcal{O}$, and hence $\neg\varphi \in D_{\mathcal{O}}(\mathfrak{N})$. Since $D_{\mathcal{O}}(\mathfrak{N}) \subset D_{\mathcal{O}}(\mathfrak{M})$, $\neg\varphi \in D_{\mathcal{O}}(\mathfrak{M})$. Thus $D_{\mathcal{O}}(\mathfrak{M})$ contains both φ and $\neg\varphi$, a contradiction. □

As seen above, when $\mathcal{O}$ is closed under negation, the notion of $\mathcal{O}$-minimal model in $\mathcal{W}$ and the notion of $\mathcal{W}$-model coincide. As a result, property 2.3 will hold if one replaces $\mathrm{Cn}_{\mathcal{W}}(.)$ by $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(.)$ and $\mathrm{Mod}_{\mathcal{W}}(.)$ by $\mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(.)$. This does not hold in general. In particular, there is not a nice relationship between the notion of $\mathcal{O}$-minimal logical consequence in $\mathcal{W}$ and the notion of $\mathcal{O}$-minimal model in $\mathcal{W}$. To see this, let's firstly consider the following proposition, which proves the "idempotence" property of $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$.

Proposition 2.10. *For all $T \subseteq \mathcal{L}$, $Cn^{\mathcal{O}}_{\mathcal{W}}(T) = Cn^{\mathcal{O}}_{\mathcal{W}}(Cn^{\mathcal{O}}_{\mathcal{W}}(T))$.*

Proof. Clearly $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T) \subseteq \mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T))$. In order to prove $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)) \subseteq \mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$ we need to prove $\mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T) \subseteq \mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T))$.

Suppose that $\mathfrak{M} \in \mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T)$, we need to show that $\mathfrak{M} \in \mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T))$. We have $\mathfrak{M} \models \mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$, i.e. $\mathfrak{M} \in \mathrm{Mod}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T))$. Moreover, if $\mathfrak{N}$ is any structure such that $\mathfrak{N} \in \mathrm{Mod}_{\mathcal{W}}(\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T))$, then $\mathfrak{N} \models T$. In other words, $\mathfrak{N} \in \mathrm{Mod}_{\mathcal{W}}(T)$. Since $\mathfrak{M} \in \mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(T)$, $D_{\mathcal{O}}(\mathfrak{N}) \not\subset D_{\mathcal{O}}(\mathfrak{M})$. Thus $\mathfrak{M}$ is an $\mathcal{O}$-minimal model of $\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$ in $\mathcal{W}$. □

The following example will show that Property 2.3 will not hold when we add the minimality condition. It will show that there are theories which have the same $\mathcal{O}$-minimal logical consequences in a class of possible worlds $\mathcal{W}$, but different sets of $\mathcal{O}$-minimal models in $\mathcal{W}$.

Example 2.11. *Let $S = \{\overline{n} \mid n \in \mathbb{N}\} \cup \{P\}$ where P is a unary predicate symbol. Let $\mathcal{O}$ be the set of all atomic sentences. Let $\mathfrak{M}_i$ and $\mathfrak{N}_i$ be the Herbrand structures defined as follows.*

- $\mathfrak{M}_i \models P\overline{n}$ *iff* $n \in \{2j \mid 0 \leq j \leq i\} \cup \{2j+1 \mid j > i\}$
- $\mathfrak{N}_i \models P\overline{n}$ *iff* $n \in \{2j \mid j \geq i\}$

Set $\mathcal{H}_1 = \{\mathfrak{M}_i \mid i \in \mathbb{N}\}$, $\mathcal{H}_2 = \{\mathfrak{N}_i \mid i \in \mathbb{N}\}$ and $\mathcal{W} = \mathcal{H}_1 \cup \mathcal{H}_2$. Then $Mod^{\mathcal{O}}_{\mathcal{W}}(\emptyset) = \mathcal{H}_1$ and $Mod^{\mathcal{O}}_{\mathcal{W}}(Cn^{\mathcal{O}}_{\mathcal{W}}(\emptyset)) = \mathcal{H}_1 \cup \{\mathfrak{N}_0\}$ and therefore $Mod^{\mathcal{O}}_{\mathcal{W}}(\emptyset) \subset Mod^{\mathcal{O}}_{\mathcal{W}}(Cn^{\mathcal{O}}_{\mathcal{W}}(\emptyset))$

The proof of this example can be found in the appendix. In this example, two theories, $\emptyset$ and $\mathrm{Mod}^{\mathcal{O}}_{\mathcal{W}}(\emptyset)$, have the same $\mathcal{O}$-minimal logical consequences in $\mathcal{W}$ (i.e., $\mathrm{Cn}_{\mathcal{W}}(P\overline{0})$). However, their sets of $\mathcal{O}$-minimal models in $\mathcal{W}$ are not the same.

On the other hand, having the same sets of $\mathcal{O}$-minimal models in $\mathcal{W}$ forces the theories to have the same $\mathcal{O}$-minimal logical consequences in $\mathcal{W}$. This statement can be made more general as in the following property.

Property 2.12. *Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be two sets of sentences, and $T, T' \in \mathcal{L}$. If $Mod^{\mathcal{O}_1}_{\mathcal{W}}(T) = Mod^{\mathcal{O}_2}_{\mathcal{W}}(T')$ then $Cn^{\mathcal{O}_1}_{\mathcal{W}}(T) = Cn^{\mathcal{O}_2}_{\mathcal{W}}(T')$.*

A number of other properties of the $\mathcal{O}$-minimality can be found in the appendix which also provide connections between $\mathcal{O}$-minimality and the *closed world assumption* ([Luk90]). These results prove the generality of the framework which are being investigated.

The next section will briefly give some remarks on the notion of hierarchy of logical consequence. One will find the structure of the $\mathcal{O}$-minimal logical consequence in $\mathcal{W}$ resembles the Borel hierarchy. It also corresponds to different level of "discovery", where level Σ_1 queries can be answered by Prolog, and some level Σ_2 queries which can be answered by a system which has been built in this thesis, and which will be explained in the next chapters.

2.3 Hierarchies of logical consequences

Given a paradigm $\mathcal{P} = (S, \mathcal{W}, \mathcal{O}, \mathcal{T})$, let T be a theory in $\mathcal{T}$. The set of $\mathcal{O}$-minimal logical consequences of T in $\mathcal{W}$ ($\mathrm{Cn}^{\mathcal{O}}_{\mathcal{W}}(T)$) has a hierarchical structure which is far more complex than the structure of $\mathrm{Cn}(T)$, Providing full description of the hierarchies of the $\mathcal{O}$-minimal logical consequences is beyond the scope of this thesis. The theory is still under development which is carried out at UNSW by the authors of [MSS]. This section aims at presenting some aspects of the theory, which will provide basic understanding of the system which is presented in the next chapters.

Basically, the $\mathcal{O}$-minimal logical consequences of a theory T are classified by the level of difficulty in inferring them from T. Sentences on higher levels in the hierarchy are derived from sentences on lower levels. The levels are either Σ's levels or Π's levels. The first level consists of members of T, and can be called either Σ_0 or Π_0. The next level is the Σ_1 level, whose members are derived from sentences in level Π_0 by a deductive step. Sentences in level Π_1 are obtained from sentences in level Σ_1 by an inductive step. Then the level Σ_2 is obtained from level Π_1 by a deductive step, and so on.

This mechanism of derivation corresponds to the alternation of compactness and weak compactness properties in the structures of $\mathcal{O}$-minimal logical consequences of T in $\mathcal{W}$. The deductive step in obtaining the Σ levels are based on a compactness property: a derivation of φ from a finite set of $\mathcal{O}$-minimal logical consequences in $\mathcal{W}$ of T guarantees that φ is an $\mathcal{O}$-minimal logical consequence of T in $\mathcal{W}$. The inductive step in obtaining the Π levels are based on a weak compactness property which specifies that φ could not belong to the set of sentences that are not $\mathcal{O}$-minimal logical consequences of T without this fact being discovered on a lower level.

It can be seen that when $\mathcal{W}$ is the set of all Herbrand structures and $\mathcal{O}$ is the set of all basic sentences then all Σ_1 (resp. Σ_2) sentences true in a possible theory T occur in the hierarchy built over T on level Σ_1 (resp. Σ_2). The Σ_1 (but not Σ_2) sentences are handled by Prolog, which has the ability to perform deduction but no ability to perform induction. On the other hand the system that has been built in this thesis has the ability to handle Σ_2 sentences. Such sentences broaden considerably the spectrum of applications. For example, a Σ_2 sentence may be used to discover the hypothesis to be learned in a learning paradigm ([OSW86]). A learning problem then becomes the problem of answering a Σ_2 query. While Σ_1 queries are discovered by a deductive step, Σ_2 queries are discovered by a deductive step followed by an inductive step. Thus a mechanism to perform both deduction and induction is necessary. This is achieved by the system *RichProlog*, which will be described in the next chapters.

Chapter 3

A Σ_2 proof procedure

Prolog can be seen as a proof procedure for the level Σ_1, where one can take $\mathcal{W}$ as the set of all Herbrand structures and $\mathcal{O}$ as the set of all atomic sentences. Another choice can be $\mathcal{W}$ equal to the set of all structures and $\mathcal{O}$ equal to any set which is closed under negation. This freedom of choice is due to the very restrictive choice of the formulas involved. Given $T \subseteq \mathcal{L}$ as a set of Horn clauses and a quantifier free formula $\psi(\overline{x})$, Prolog determines whether or not $T \models^{\mathcal{O}}_{\mathcal{W}} \exists\overline{x}\psi(\overline{x})$. If the answer is yes, it outputs a sequence of terms $\overline{t}$ such that $T \models^{\mathcal{O}}_{\mathcal{W}} \psi(\overline{t})$.

In this thesis, a proof procedure for the level Σ_2, *RichProlog*, has been built. $\mathcal{W}$ is taken as the set of all Herbrand structures, while $\mathcal{O}$ is taken as the set of atomic/basic sentences (corresponding to positive/positive and negative data). The procedure will determine, for an atomic formula $\psi(\overline{x}, \overline{y})$, whether or not $T \models^{\mathcal{O}}_{\mathcal{W}} \exists\overline{x}\forall\overline{y}\psi(\overline{x}, \overline{y})$. If the answer is yes, it outputs a sequence of terms $\overline{t}$ (a *witness* or *hypothesis* for $\overline{x}$) such that $T \models^{\mathcal{O}}_{\mathcal{W}} \forall\overline{y}\psi(\overline{t}, \overline{y})$. An algorithm for this proof procedure has been outlined in [MSS]. More complex form for the Σ_2 sentence $\exists\overline{x}\forall\overline{y}\psi(\overline{x}, \overline{y})$ (i.e., the case where $\psi(\overline{x}, \overline{y})$ is not atomic) can be transformed to the above form. The system that has been implemented in this thesis follows a slightly modified version of the algorithm given in [MSS]. The difference will be discussed in 3.1.

This chapter describes the algorithms and the related issues. The next chapter will present the details of the implementation. Layout of this chapter is as follows. The first section discusses the algorithms, as well as some unique features of the system that do not appear in an ordinary Prolog system, such as handling formulas in Π_1 level, the process of building logic programs and the transformation of queries. It also discusses handling of true negation, as opposed to negation as failure. The second section illustrates the algorithms by giving an example. A set of clauses will be given, and then different queries will be carried out. The next section addresses some characteristics in the backtracking mechanism in *RichProlog*. Finally, the last section discusses some remaining problems that affect the current implementation of *RichProlog*.

3.1 The algorithms

In this section, two (slightly different) algorithms will be presented. These are the nondeterministic algorithms for handling query in level Σ_2. Their actual implementations require some concepts, which will be introduced following the algorithms.

In the following algorithms, we assume that there is a measure μ over the set of all closed terms, extended to a measure over the set of all terms by $\mu(t) = \mu(\{\mu(t') \mid t' \text{ is a closed instance of } t\})$. The m-product of μ is also denoted μ. Assume also that some enumeration of all m-tuples of closed terms is given.

3.1.1 The first algorithm

The first algorithm has been discussed in [MSS]. It is presented here for convenience (Figure 3.1). This algorithm, given an enumeration of the closed instances of the universally quantified variables $y_1, \ldots, y_m$, will (nondeterministically) output the hypothesis for the existentially quantified variables.

We will now define some terminology before going to further discussion. The final value of $(t_1, \ldots, t_n)$, as it is at the end of step 6, is a *candidate for the hypothesis* (or just *candidate* for short). It is said to be *generated* by $(t'_1, \ldots, t'_m)$ and also to *cover* $(t'_1, \ldots, t'_m)$. Sometimes the value of $(t_1, \ldots, t_n)$ from the previous loop is of concern. In that case, we might say the value of $(t_1, \ldots, t_n)$, as at the end of step 6, is *generated* from the previous value of $(t_1, \ldots, t_n)$ by $(t'_1, \ldots, t'_m)$.

In this algorithm, the first candidate is initialized to the most general instance of the existentially quantified variables. At each iteration of the outermost while loop, a member of the enumeration that has not been covered is chosen to generate a new candidate from the previous one. Thus the candidate $(t_1, \ldots, t_n)$ becomes more specific after each iteration of the outermost while loop. The loop stops when the measure of the set Y reaches some predetermined *threshold* (or *degree of confirmation*) δ.

It can be seen that the search for the unifiers $\theta_0 \ldots \theta_k$ in step 3 would be more oriented if the variables $(y_1, \ldots, y_m)$ are assigned the corresponding values of the closed term from the enumeration. The second algorithm presented in the next part exploits this feature.

3.1.2 The second algorithm

In this algorithm (Figure 3.2), the search for candidates is guided by sequentially assigning the closed terms from the enumeration to the universal variables appearing in the query.

Input: A finite logic program T, a sentence φ of the form $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \psi$ where ψ is atomic, and a rational number δ in $(0,1)$.
Output: A hypothesis for $(x_1, \ldots, x_n)$.
Initialize Y to $\emptyset$.
Initialize $(t_1, \ldots, t_n)$ to $(x_1, \ldots, x_n)$.
While $\mu(Y) < \delta$ *do*

1. $Res = \{\psi\}$, $k = 0$.
2. $(t'_1, \ldots, t'_m)$ = the first member of the enumeration of m-tuples of closed terms that is not an instance of a member of Y.
3. *While* $Res \neq \emptyset$ *do*
   - 3.1. Choose ρ in Res and (renamed) clause $\alpha_0 \leftarrow \alpha_1 \ldots \alpha_p$ in T such that
     - ρ and α_0 unify with mgu θ_k;
     - $(t'_1, \ldots, t'_m)$ is an instance of $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k$.

     (If no such ρ and $\alpha_0, \ldots, \alpha_p$ exist, exit 3.)
   - 3.2 Replace ρ by $\alpha_1, \ldots, \alpha_p$ in Res.
   - 3.3 Apply θ_k to Res.
   - 3.4 $k = k + 1$.
4. *If* $Res \neq \emptyset$ *then* output *no*.
5. *Else* let θ be most general such that:
   - no variable occurs both in $(t_1, \ldots, t_n)\theta_0 \ldots \theta_k\theta$ and $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta$;
   - $(t'_1, \ldots, t'_m)$ is an instance of $(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta$.
6. $(t_1, \ldots, t_n) = (t_1, \ldots, t_n)\theta_0 \ldots \theta_k\theta$.
7. $Y = Y \cup \{(y_1, \ldots, y_m)\theta_0 \ldots \theta_k\theta\}$.

Output $(t_1, \ldots, t_n)$.

Figure 3.1: The nondeterministic algorithm for a Σ_2 proof procedure, as outlined in [MSS]

The nondeterministic choice in step 3.1 of both algorithms needs to be simulated when one attempts to implement their deterministic equivalence. This is where more differences between the algorithms are observed. To eliminate nondeterminism, one would use sequential search with backtracking (discussed in section 3.3). The differences in both algorithms can be seen as follows. In order to generate a new candidate, the search in the first algorithm follows the exact path that it has already gone through, passing and then going beyond the current point. The search in second algorithm, on the other hand, will jump to possible branch and start from a new position.

During the sequential search (at level Σ_2), *RichProlog* needs to validate its hypothesis. This happens when the set Y has measure greater than or equal to the degree of confirmation. It involves handling Π_1 formulas, which has been thoroughly discussed in [MSS]. In order

Input: A finite logic program T, a sentence φ of the form $\exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m \psi$ where ψ is atomic, and a rational number δ in $(0,1)$.
Output: A hypothesis for $(x_1, \ldots, x_n)$.
Initialize Y to $\emptyset$.
Initialize $(t_1, \ldots, t_n)$ to $(x_1, \ldots, x_n)$.
While $\mu(Y) < \delta$ *do*

1. $Res = \{\psi\}$, $k = 0$.
2. $(t'_1, \ldots, t'_m)$ = the first member of the enumeration of m-tuples of closed terms that is not an instance of a member of Y.
3. *While* $Res \neq \emptyset$ *do*
   - 3.1. Choose ρ in Res and (renamed) clause $\alpha_0 \leftarrow \alpha_1 \ldots \alpha_p$ in T such that ρ and α_0 unify with mgu θ_k; (If no such ρ and $\alpha_0, \ldots, \alpha_p$ exist, exit 3.)
   - 3.2 Replace ρ by $\alpha_1, \ldots, \alpha_p$ in Res.
   - 3.3 Apply θ_k to Res.
   - 3.4 $k = k + 1$.
4. *If* $Res \neq \emptyset$ *then* output *no*.
5. $(t_1, \ldots, t_n) = (t_1, \ldots, t_n)\theta_0 \ldots \theta_k \theta$.
6. $Y = Y \cup \{(y_1, \ldots, y_m)\theta_0 \ldots \theta_k \theta\}$.

Output $(t_1, \ldots, t_n)$.

Figure 3.2: Another nondeterministic algorithm for a Σ_2 proof procedure

to provide a complete picture of the system, it will be summarized in the next part. Before going into details of how to handle Π_1 formulas, some terminology which is needed in the following sections will be introduced.

A *clause* is defined as a set of basic formulas. A (generalized) *rule* is a formula of the form φ_0 or $\varphi_1 \wedge \ldots \wedge \varphi_n \rightarrow \varphi_0$, where $\varphi_0, \ldots, \varphi_n$ are basic formulas. φ_0 is called the *head*, while $\varphi_1 \wedge \ldots \wedge \varphi_n$ (in the second case) is called the *body* of the rule. Note that classical logic programs consider only definite Horn clauses and rules where the head and all formulas in the body (if any) are atomic formulas. In that case, both representations are equivalent. In general, there are n rules associated with a clause of cardinality n. The following example has been given in [MSS]. Consider the clause $\{p, q, \neg r\}$, the rules associated with it are $p \leftarrow \neg q \wedge r$, $q \leftarrow r \wedge \neg p$ and $\neg r \leftarrow \neg p \wedge \neg q$. Note that by this definition, *fact* is a special type of rule, with the head being the fact itself. A (generalized) logic program T (i.e., a set of generalized rules) is *derived* from a set $\mathcal{S}$ of clauses if every rule in T is associated with a clause in $\mathcal{S}$.

3.1.3 Handling Π_1 formulas

While Σ_k formulas are obtained by a deductive step, Π_k formulas are obtained by an inductive step. Consider a Π_1 formula φ of the form $\forall y \psi[t_1/x_1, \ldots, t_n/x_n]$. It belongs to level Π_1 on the hierarchy over a theory T if it is *refutable*, i.e.,

$$\text{if } T' \not\models^{\mathcal{O}}_{\mathcal{W}} \forall y \psi[t_1/x_1, \ldots, t_n/x_n] \text{ then } T' \models^{\mathcal{O}}_{\mathcal{W}} \neg\forall y \psi[t_1/x_1, \ldots, t_n/x_n].$$

Let $\chi = \psi[t_1/x_1, \ldots, t_n/x_n]$, then $\neg\forall y\chi$ is transformed into $\exists y \neg\chi$. To check that $T \not\models^{\mathcal{O}}_{\mathcal{W}} \exists y \neg\chi$, unification can be used. (Note that at this point, if the check returns negative result (i.e., $T \models^{\mathcal{O}}_{\mathcal{W}} \exists y \neg\chi$), then the computation will backtrack to level Σ_2. More details of this can be found in 3.3). Basically this involves the generalization of logic program. It requires the introduction of *true negation* and the process of building programs that are determined by the (Σ_2) queries. The following parts contain discussions on these matters.

3.1.4 True negation

A Π_1 query $\forall y\chi$ is handled by refutation. Thus there is a need to perform classical SLD-Resolution with queries and programs containing both negative and positive formulas. The notion of negation here can be termed *true negation*. This is different from the notion of negation in Prolog, where it is implemented as *failure*, i.e.,

$$\textit{not } X \leftarrow X \text{ is not provable}$$

The difference can be observed, for example, in the case where we need to explicitly specify negative facts. One may wish to specify that $p(a), \neg p(b)$ are all the facts which are known about the property p. This cannot be achieved if negation is implemented as failure. Note that introducing new symbols to stand for negation (for example *dislike* to stand for $\neg$*like*) would mean expanding the vocabulary, i.e., changing the paradigm that we are working on. Moreover, the new symbols then need to be interpreted, but the interpretation may have nothing to do with the logic itself. In the example above, the English meaning suggests that *dislike* be interpreted as $\neg$*like*. However, it would be permissible to have *dislike* $\wedge$ *like*.

Basically, one needs to negate literals when building the logic programs for the query and its negation. This is investigated in the next part.

3.1.5 Building logic programs

How to build a logic program from a query and a set $\mathcal{S}$ of clauses has been given in [MSS]. It can be summarized as follows (Figure 3.3). Note that during the building process, a *conflict* is said to be detected if there are atomic formulas ψ and χ such that both ψ and $\neg\chi$ are in T, and ψ and χ unify. In that case, the building process will stop and signal that the logic program cannot be built.

Input: A set $\mathcal{S}$ of clauses, a sentence φ
Output:

- $False$ if there is conflict in logic program derived from φ and $\mathcal{S}$
- Otherwise outputs $True$ and the derived program.

Initialize T to $\emptyset$.
Initialize Γ to the set of basic formulas in φ
if Γ contains ψ and $\neg\chi$ such that ψ and χ unify *then* output $False$
For each element of Γ, set a flag indicating them as *not_checked*
While there are elements of Γ with the flag set to *not_checked*

- Let α be an element of Γ which has the *not_checked* flag
- $T = T \cup \{$rules associated with clauses in $\mathcal{S}$ whose head unifies with $\alpha\}$
- $X = X \cup \{$formulas in the body of the new rules in $T\}$
- *if* Γ contains ψ and $\neg\chi$ such that ψ and χ unify *then* output $False$
- set the flag of α as *checked*

Output $True$ and T.

Figure 3.3: Algorithm for building a logic program

Given a set $\mathcal{S}$ of clauses, the logic programs derived from different generalized queries are generally different. In particular, one clause may be used to produce different rules in different logic programs. The searching process, therefore, will be more directed if separate logic programs are built for different queries. Also, a logic program generally does not contain all rules associated with a clause. Thus delaying the building of logic programs until a query is given has the advantage of saving space which might be used to store rules that will never be used. This is obviously at the cost of a negligible time spending in building logic programs before the proof procedure can be carried out. More importantly, building programs helps to detect possible conflicts. As can be seen above, validating a hypothesis will require a different logic program from the one used in generating it. Merging two logic programs is possible. However, in order to detect conflicts within each program, it is necessary that each of them is built separately.

3.1.6 Transforming queries

The discussions so far have considered a Σ_2 query whose matrix is an atomic formula. It has been shown that complex queries can be transformed into queries of this form ([MSS]).

Basically, a new predicate symbol is introduced for each inner node in the parse tree of the matrix of the query. Note that these symbols are not part of the vocabulary. They can be considered as artificial symbols that only appear in a program derived from a particular query. Once the query has been handled, they no longer exist.

Each of these predicates should have arity equal to the total number of existential and universal variables in the query. The query's matrix itself is replaced by the new predicate assigned to the root of the parse tree. Figure 3.4 depicts the parse tree for the matrix of the query $\varphi = \exists \mathtt{X} \exists \mathtt{Y} \forall \mathtt{Z} \psi$ where

$$\psi = \mathtt{p}_1(\mathtt{X},\mathtt{Y}) \wedge [\neg \mathtt{p}_2(\mathtt{Z}) \vee (\mathtt{p}_3(\mathtt{X},\mathtt{Y}) \wedge \mathtt{p}_4(\mathtt{Y}) \wedge (\neg \mathtt{p}_2(\mathtt{Y}) \vee \neg \mathtt{p}_3(\mathtt{X},\mathtt{Z})))]$$

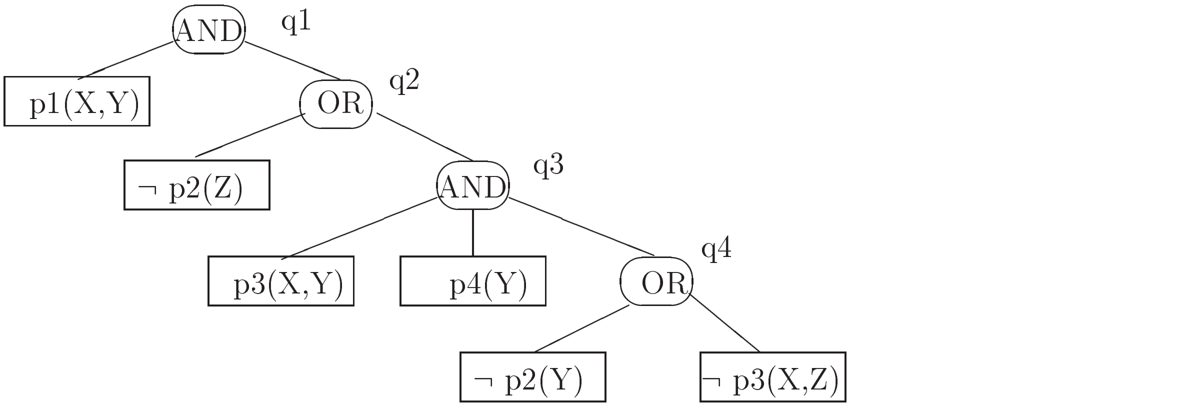

Figure 3.4: Parse tree for $\mathtt{p}_1(\mathtt{X},\mathtt{Y}) \wedge [\neg \mathtt{p}_2(\mathtt{Z}) \vee (\mathtt{p}_3(\mathtt{X},\mathtt{Y}) \wedge \mathtt{p}_4(\mathtt{Y}) \wedge (\neg \mathtt{p}_2(\mathtt{Y}) \vee \neg \mathtt{p}_3(\mathtt{X},\mathtt{Z})))]$

Note that it is not necessary to introduce new predicate symbols for the leaves of the parse tree. In this example, 4 new predicates have been created. Additional rules are created as the following will show. Note that similar to the situation for the "artificial" predicates, these rules are not part of the set $\mathcal{S}$ (which is a set of clauses). They are the rules created in the transformation of a particular query, and are used in the proof for that query only. They will no longer exist once the proof is done.

To make the following statement general, the term "the predicate at a node" will generally mean the new predicate introduced to the node if it is an inner node, otherwise the predicate at the leaf node itself. Basically, if an inner node is an *and-node*, then a new rule is created with the head being the predicate at that node, and the body being the conjunction of the predicates at the children of the node. On the other hand, if the inner node is an *or-node*, then for each child of the node, one new rule is added to the program. The head of the new rule is the predicate at the parent node, while the body is the predicate at the child node. In this example, the additional rules are:

$$\mathtt{q}_1(\mathtt{X},\mathtt{Y},\mathtt{Z}) \quad \leftarrow \quad \mathtt{p}_1(\mathtt{X},\mathtt{Y}), \mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z})$$

$$
\begin{aligned}
\mathtt{q_2(X,Y,Z)} &\leftarrow \neg\mathtt{p_2(Z)}\\
\mathtt{q_2(X,Y,Z)} &\leftarrow \mathtt{q_3(X,Y,Z)}\\
\mathtt{q_3(X,Y,Z)} &\leftarrow \mathtt{p_3(X,Y),p_4(Y),q_4(X,Y,Z)}\\
\mathtt{q_4(X,Y,Z)} &\leftarrow \neg\mathtt{p_2(Y)}\\
\mathtt{q_4(X,Y,Z)} &\leftarrow \neg\mathtt{p_3(X,Z)}
\end{aligned}
$$

The query will be transformed as $\exists\mathtt{X}\exists\mathtt{Y}\forall\mathtt{Z}\ \mathtt{q_1(X,Y,Z)}$

3.2 An example

To illustrate the arguments so far, we now consider a simple example about ordering of elements in a (possibly infinite) set (Figure 3.5). In this example, `predecessor(X,Y)` is true if `X` is a predecessor of `Y`. Similarly, `successor(X,Y)` is true if tt X is a successor of `Y`. The predicates `equal`, `smaller`, `smaller_eq`, `larger`, `larger_eq` stand for $=, <, \leq, >, \geq$ correspondingly.

1. smaller(X,Y) :- predecessor(X,Y).
2. smaller(X,Y) :- predecessor(X,Z), smaller(Z,Y).
3. ¬ smaller(X,X).
4. smaller_eq(X,X).
5. smaller_eq(X,Y) :- predecessor(X,Z), smaller_eq(Z,Y).
6. ¬ smaller_eq(X,Y) :- predecessor(Y,X).
7. successor(X,Y) :- predecessor(Y,X).
8. larger(X,Y) :- successor(X,Y).
9. larger(X,Y) :- successor(X,Z), larger(Z,Y).
10. ¬ larger(X,X).
11. larger_eq(X,X).
12. larger_eq(X,Y) :- successor(X,Z), larger_eq(Z,Y).
13. ¬ larger_eq(X,Y) :- successor(Y,X).
14. equal(X,X).

15. predecessor(a,b).
16. predecessor(b,c).
17. predecessor(c,d).
18. predecessor(d,e).

Figure 3.5: The "order" example

Figure 3.5 gives details of a set $\mathcal{S}$ of clauses. We now proceed with different queries. For each query, the first step is to build the logic programs. If conflicts are detected during this step, *RichProlog* will announce that it is unable to answer the query. Once the programs are built, the second algorithm (Figure 3.2) will be applied to find the hypothesis.

3.2.1 A simple query

This simple query is meant to find the least number in the set. We will show the sets Γ's as they arise during the process of building logic programs. We also show the programs derived from the query and the set $\mathcal{S}$.

$$\varphi = \exists \mathtt{X} \forall \mathtt{Y}\ \mathtt{smaller\_eq(X,Y)}$$

Following the algorithm given in Figure 3.3 to build the program for the query, we have:

The set Γ_1 is

{

`smaller_eq(X,Y)`, (from the query φ)

`predecessor(X,Z)`, (from 5.)

`smaller_eq(Z,Y)` (from 5.)

}

The program T_1 derived from the query and $\mathcal{S}$ is

`smaller_eq(X,X)`. (4.)

`smaller_eq(X,Y) :- predecessor(X,Z), smaller_eq(Z,Y)`. (5.)

`predecessor(a,b)`. (15.)

`predecessor(b,c)`. (16.)

`predecessor(c,d)`. (17.)

`predecessor(d,e)`. (18.)

Now the query is negated and another program is built for its negation.

The set Γ_2 is

{

¬ `smaller_eq(X,Y)`, (from ¬φ)

`predecessor(X,Z)`, (from 5.)

¬ `smaller_eq(X,Y)`, (from 5.)

`predecessor(Y,X)` (from 6.)

}

The program T_2 derived from the ¬φ and $\mathcal{S}$ is

¬`smaller_eq(Z,Y) :- predecessor(X,Z),` ¬ `smaller_eq(X,Y)`. (5.)

¬`smaller_eq(X,Y) :- predecessor(Y,X)`. (6.)

`predecessor(a,b)`. (15.)

`predecessor(b,c)`. (16.)

`predecessor(c,d)`. (17.)

$\texttt{predecessor(d, e)}$. (18.)

There is no conflict detected during the process of building programs. There is no need to transform the query in this case. Assume that an enumeration of closed terms `a, b, c, d, e` is available. Each of these has measure 0.15, and the degree of confirmation is 0.7. Running the nondeterministic algorithm given in Figure 3.2 with the query φ and the program T_1 successfully outputs the hypothesis `X = a`. Note that the nondeterministic algorithm does not require the second program T_2. Building this program is needed, however, in order to detect the conflict, and to test the validity of the hypothesis in the deterministic implementation of the algorithm.

The next 2 queries are intended to find the first three members of the set. They are not optimal, and are presented only to demonstrate the process of building logic programs, and to show an example where *RichProlog* refuses to answer a query.

3.2.2 A query that needs transformation

The following query is more complex than the previous one. It will need to be transformed. *RichProlog* should be able to handle it, since there will be no conflict in the derived programs.

$\varphi = \exists \mathtt{X} \exists \mathtt{Y} \exists \mathtt{Z} \forall \mathtt{W}[\mathtt{smaller}(\mathtt{X},\mathtt{Y}) \wedge \mathtt{smaller}(\mathtt{Y},\mathtt{Z}) \wedge (\mathtt{equal}(\mathtt{X},\mathtt{W}) \vee \mathtt{equal}(\mathtt{Y},\mathtt{W}) \vee \mathtt{smaller\_eq}(\mathtt{Z},\mathtt{W}))]$

Two programs will be built as follows.

The set Γ_1 is

{

smaller(X, Y), (from the query φ)

smaller(Y, Z), (from the query φ)

equal(X, W), (from the query φ)

equal(Y, W), (from the query φ)

smaller_eq(Z, W), (from the query φ)

predecessor(X, Y), (from 1.)

predecessor(X, Z), (from 2.)

smaller(Z, Y), (from 2.)

predecessor(X, Z), (from 5.)

smaller_eq(Z, Y) (from 5.)

}

The program T_1 derived from the query and $\mathcal{S}$ is

smaller(X, Y) :- predecessor(X, Y). (1.)

smaller(X, Y) :- predecessor(X, Z), smaller(Z, Y). (2.)

equal(X, X). (14.)
smaller_eq(X, X). (4.)
smaller_eq(X, Y) :- predecessor(X, Z), smaller_eq(Z, Y).(5.)
predecessor(a, b). (15.)
predecessor(b, c). (16.)
predecessor(c, d). (17.)
predecessor(d, e). (18.)

Another logic program is derived from the query and the set $\mathcal{S}$:

The set Γ_2 is
{
¬smaller(X, Y), (from $\neg\varphi$)
¬smaller(Y, Z), (from $\neg\varphi$)
¬equal(X, W), (from $\neg\varphi$)
¬equal(Y, W), (from $\neg\varphi$)
¬smaller_eq(Z, W), (from $\neg\varphi$)
predecessor(X, Z), (from 2.)
¬smaller(X, Y), (from 2.)
predecessor(X, Z), (from 5.)
¬smaller_eq(X, Y), (from 5.)
predecessor(Y, Z) (from 6.)
}

The program T_2:
¬smaller(Z, Y) :- predecessor(X, Z), ¬smaller(X, Y). (2.)
¬smaller(X, X). (3.)
predecessor(a, b). (15.)
predecessor(b, c). (16.)
predecessor(c, d). (17.)
predecessor(d, e). (18.)

Now the query and its negation are transformed. This introduces temporary predicates and rules into the logic programs.

Additional rules for program T_1 are:
q_1(X, Y, Z, W) :- smaller(X, Y) $\wedge$ smaller(Y, Z) $\wedge$ q_2(X, Y, Z, W).
q_2(X, Y, Z, W) :- equal(X, W).
q_2(X, Y, Z, W) :- equal(Y, w).

$\mathtt{q_2(X,Y,Z,W)}$:- $\mathtt{smaller\_eq(Z,w)}$.

The Σ_2 formula that is being concerned is $\exists \mathtt{X} \exists \mathtt{Y} \exists \mathtt{Z} \forall \mathtt{W}\ \mathtt{q_1(X,Y,Z,W)}$.

Additional rules for program T_2 are:
$\mathtt{q_3(X,Y,Z,W)}$:- $\neg\mathtt{smaller(X,Y)}$.
$\mathtt{q_3(X,Y,Z,W)}$:- $\neg\mathtt{smaller(Y,Z)}$.
$\mathtt{q_3(X,Y,Z,W)}$:- $\mathtt{q_4(X,Y,Z,W)}$.
$\mathtt{q_4(X,Y,Z,W)}$:- $\neg\mathtt{equal(X,W)} \land \neg\mathtt{equal(Y,W)} \land \neg\mathtt{smaller\_eq(Z,W)}$.

The Σ_1 formula that is being concerned is $\mathtt{q_3(X,Y,Z,W)}$.

For this query, there is no conflict detected. The next query, intended to return the same results, however, is not considered feasible.

3.2.3 A query that cannot be answered

The query considered here also intends to find the 3 smallest members in the set. However, the programs built from its negation and the set $\mathcal{S}$ contains conflicts.

$\varphi = \exists \mathtt{X} \exists \mathtt{Y} \exists \mathtt{Z} \forall \mathtt{W} [\mathtt{predecessor(X,Y)} \land \mathtt{predecessor(Y,Z)} \land (\mathtt{equal(X,W)} \lor \mathtt{equal(Y,W)} \lor \mathtt{smaller\_eq(Z,W)})]$

The following set Γ_2 is obtained when during the process of building the logic program for $\neg\varphi$:

{
$\neg\mathtt{predecessor(X,Y)}$, (from $\neg\varphi$)
$\neg\mathtt{predecessor(Y,Z)}$, (from $\neg\varphi$)
$\neg\mathtt{equal(X,W)}$, (from $\neg\varphi$)
$\neg\mathtt{equal(Y,W)}$, (from $\neg\varphi$)
$\neg\mathtt{smaller\_eq(Z,W)}$, (from $\neg\varphi$)
$\mathtt{predecessor(X,Z)}$, (from 2.)
...
}

Γ_2 contains $\neg\mathtt{predecessor(X,Y)}$ and $\mathtt{predecessor(X,Z)}$. The query is therefore not attempted.

3.3 Backtracking

In Prolog, answers to a query are obtained by searching for refutation of the negation of the query. This is done using a particular strategy of resolution called SLD-resolution. Note that since the clauses are Horn clauses, in the actual implementation of Prolog, formulas are never negated. One is referred to [SS97] and [Boi93] for detailed discussions on searching and backtracking in a Prolog interpreter. Essentially, the search involves traversing (in depth first manner) a *search tree*, backtracking on encountering a *failure node*. Results of the search can be printed out at each of the *success nodes*. In general, a search tree may have multiple success nodes. It may also have *infinite branches*.

In *RichProlog*, the search procedure is complicated by a number of factors. Following the second algorithm (3.1.2), it iterates through a number of *stages*, corresponding to loops of the outermost while loop. Generating a candidate from the previous one at each stage is done by SLD-resolution as employed by Prolog interpreters. Each branch in the search tree traversed by *RichProlog*, therefore, can be viewed as consisting of a number of stages, each of them being a branch in a normal Prolog search tree. The normal Prolog search tree is said to be *associated* with that stage. A stage starts when the search for one candidate starts, and ends when a candidate is generated. If the measure of the set Y is below the threshold, advancing to the next stage happens when a success node of the search tree in current stage is encountered. A hypothesis is generated when a success node is found and the measure of Y reaches the threshold. At this point, it is necessary to carry out the test for the validity of the hypothesis.

Another issue, therefore, involves different treatments of backtracking in a search tree associated with each stage, and backtracking between stages. Negative result from the test for validity of the hypothesis causes a backtrack in the last stage. In general, within one stage, negative result in the last branch of the search tree associated with that stage causes a backtrack to the previous stage in order to find another candidate. Negative result in the last branch of the search tree associated with the first stage means negative answer to the query.

Note that the search tree associated with a stage may contain infinite branches. In that case, the search for a candidate may never terminate. The program for negation of the query may also encounter an infinite loop. However, in this case, a hypothesis is already stated, and as long as the validation of the hypothesis is still in progress, it is considered to be a final hypothesis output by the system. More details of how the search in level Σ_2 is performed in *RichProlog* can be found in the next chapter (section 4.3)

3.4 Issues in designing the set of clauses

As discussed in [SS97], the rule order and goal order in a logic program play important roles in the execution of that program. They become much more significant in *RichProlog*, due to the fact that more than one rule can be associated with a clause. Yet another problem that can be solved in Prolog, but not in *RichProlog*, is the use of recursive programming in the application level. This section will address these issues and the solutions that are used in current *RichProlog*.

3.4.1 Goal order

Consider the example given in section 3.2 again. The following 2 rules are equivalent, and both are associated with the clause 5.

$$\neg\ \texttt{smaller\_eq(Z,Y)} \text{ :- } \texttt{predecessor(X,Z)}, \neg\ \texttt{smaller\_eq(X,Y)}.$$

and

$$\neg\ \texttt{smaller\_eq(Z,Y)} \text{ :- } \neg\ \texttt{smaller\_eq(X,Y)}, \texttt{predecessor(X,Z)}.$$

The first rule above, together with clause 6. (Figure 3.5) will be necessary and sufficient to provide an answer to the goal ¬ `smaller_eq(a,b)`? (which is created in the validation of the hypothesis `a` for the first query, part 3.2.1). However, the second rule contains a *left recursion* definition for ¬`smaller_eq`. It will create an infinite loop for such goal. Thus choosing one of these rules is a decision must be made by the system. A solution is to swap the head and the literal in the body of the clause, together with changing their signs.

3.4.2 Rule order

Another issue is the order of input clauses. In a Prolog program, changing rule order means permuting the search tree. In particular, this results in different output order for the search results. In *RichProlog*, changing rule order will result in different order in which candidates are generated. It can also affect the efficiency of the validation of the hypothesis. Remember that the test for validity of the hypothesis can stop immediately when a counter-example for the hypothesis is found. Permuting the search tree (for the program T_2) therefore has deep impacts on the performance of the validation process.

3.4.3 Recursive clauses

Although recursion is an essential component of logic programming, its use is limited for programs in *RichProlog*. As an example, consider the following definitions of the predicate `number`, which are intended to generate natural numbers:

```
number(z).
number(s(X)) :- number(X).
```

The second clause has the following rule associated with it:

$\neg$`number(X)` :- $\neg$`number(s(X))`.

A call to $\neg$ `number(X)` may result in an infinite loop without success.

Another problem with recursive clauses is that they are too "restrictive". Consider the clause 2. (Figure 3.5) in the example given in section 3.2. A program might never be allowed to contain $\neg$`predecessor(X, Y)`. This is because the following rule, which is associated with clause 2., contains a conflict:

$\neg$`predecessor(X, Z)` :- `smaller(Z, Y)`, $\neg$`smaller(X, Y)`.

Having identified these problems, current *RichProlog* is developed in the direction of providing two working modes, in one mode, all the processes of building logic programs and transformation of queries are carried out automatically. In another, logic programs and queries transformation are left to the application. The next chapter will discuss these in detail.

Chapter 4

Implementation

This chapter describes the implementation of a system for the proof procedure that follows the second algorithm presented in the previous chapter. Taking into account the issues mentioned in the previous chapter (section 3.4), the current system aims at providing two working modes, namely *full mode* and *partial mode*. In the full mode, building logic programs and transforming queries are performed automatically by the system. In the partial mode, the logic programs and the result of query transformation are assumed to be provided by the application. The system will still check for possible conflicts in the given logic programs. Due to time constraint, the query transformation mechanism in the full mode has not been completed. This will be continued in the future work.

This chapter is organized as follows. The first section briefly reviews the implementation of an ordinary Prolog interpreter which is used as a starting point for the implementation of the system. The second section will present an example of the search tree that is present in the execution of *RichProlog*. Then the next section discusses the system in detail. An application (pattern matching) will be given in the last section.

4.1 *iProlog*

A Prolog system can be considered as a theorem-proving system where SLD-resolution is employed to find answer to a query (i.e. refutation for the negation of the query). This resolution is carried out using unification, which is a process of determining whether or not two terms can be made identical by appropriate substitutions for their variables. Prolog deals with programs in the form of a set of Horn clauses, and queries of the form $\exists \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{m} \varphi$ where φ is a conjunction of atomic formulas. In the search for the answer, a common computation strategy is to select the left-most literal in the current resolvent, while the search strategy can be trying the clauses in their order of appearance. (Note that in Prolog, a clause has only one rule associated to it, thus they can be used exchangeably. This will not be the case when we discuss *RichProlog*.) A Prolog control can then be represented by a depth-first search from left to right on an *and/or tree*. More details about Prolog can be found, for example in [Boi93, SS97].

iProlog (`http://www.cse.unsw.edu.au/~claude/research/prolog.html`) is an implementation of Prolog. It has been used as a starting point for the implementation of *RichProlog*. This section will discuss some of its features which are relevant to the implementation of *RichProlog*. More details about *iProlog* can be found in [SS83a] and [SS83b].

In this section, a *term* can be classified as *variable*, *constant symbol* or *functional term*. Functional terms are of the form

$$\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n})$$

where $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n}$ are terms. $\mathtt{f}$ is called the *functor*, $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n}$ are called the *arguments*, and $\mathtt{n}$ is called the *arity* of $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n})$. We can regard a constant symbol as a functional term of arity 0, with the functor being itself. This will provide generality for the following discussion. Terms are stored as *records*. Thus there is a record for $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n})$ and one record for each of $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{n}$. In this part of the thesis, we will try to use these terminology only. It should be noted, however, that alternatives are present in the literature. Constant symbols are sometimes referred to as *atoms* or *atomic terms*. Functional terms are also named *compound terms*, while *functional name* or *principal term* are equivalent names for functor.

4.1.1 Some remarks on data structure in *iProlog*

A term can be seen as having tree structure with the root node contains its functor. Terms which appear in a functional term can be represented as the subtrees of the tree for that term. The functors of these terms are the nodes (constants are the leaves, while other functors are inner nodes). The arcs in the tree are implemented as pointers from the record of a functional term to the records of its arguments. Records can be of different types, specified by the field *type*.

Rules whose heads have the same functor are grouped and stored in a linked list, accessed by obtaining the (common) functor of the head (note that the head of a fact is the fact itself).

Each variable contains a string for its name, as well as an offset in a *frame* such that different frames will give different bindings for a variable. More details about frames can be found in the discussion about memory management in *iProlog* (4.1.2).

Following the characterization given in [Boi93], *iProlog* belongs to the category known as *structure copying*. This means that in the unification process, new instances of terms are constructed as they become the values of some variables for the first time. For example, suppose that `f(X,h(Y))` and `f(h(g(a)), X)` are to be unified. An instance of `h(g(a))` will be constructed, as it is *accessed* for the first time when `X` and `h(g(a))` unify. When `h(Y)` and `X` unify, `Y` will be bound to `g(a)`, which is a component of the already constructed term `h(g(a))`. Using structure copying requires a special region in the memory for storing the instances of terms as they are constructed. This is discussed in the next part.

4.1.2 Some remarks on stack and memory management in *iProlog*

Variables bindings are stored in a special memory region called *global stack*. This memory area has a *copy stack or heap* behavior ([Boi93]), since its management resembles that of both stack and heap. In particular, each allocation made in this region is placed at its top, like in case of a stack. Upon backtracking, the allocated memory is restored by simply restoring the register designating its top.

As discussed earlier, each variable has an offset in the local environment called frame. The frame is used as a lookup table to access the values of the variables. Before attempting unification between a term and the head of a clause, a vector of size equal to the number of variables in that clause is allocated from a dedicated memory area called the *local stack*. This stack has similar behavior to that of the global stack as described above. The variables' offsets give access to the pointers of their values in the global stack.

Another stack present in *iProlog* is the *trail*. It is introduced as a solution to the *post-bindings* problem. Detailed discussion of the trail can be found in [Boi93].

Apart from these, *iProlog* uses a C's stack to store environment records in each call to prove and to handle backtracking. This stack stores the state of the procedure before each call to prove, so that they can be restored upon backtracking. This includes the current goal, the stack pointers for the global stack and local stack, as well as the pointer to the trail.

Other features of *iProlog* can be found in [SS83a]. In fact, *iProlog* agrees in many aspects with the discussion found in [Boi93]. These include the handling of local environment and the continuation method of controlling the proof. Instead of using structure copying, a number of Prolog systems use *structure sharing*. Differences between them can also be found in [Boi93].

4.2 An example

As described in section 3.3, the search for a hypothesis in *RichProlog* consists of a number of stages, each of them being a search for a candidate. Each stage then can be considered as a search for solution of a Σ_1 query. Thus instead of printing out the result of resolution at a success node, the proof procedure continues by assigning the next m-tuple to the universal variables and making a copy of the existential variables. Note that this copying is necessary for restoring the bindings of these variables upon backtracking from failure of subsequent stages. The following example is presented with the purpose of giving more insight into the execution of a Σ_2 search procedure.

Consider the example where the query is $\exists \texttt{X Y} \forall \texttt{Z W p(X, Y, Z, W)}$?. Assuming that the enumeration of the closed terms is a, b, f(b), g(a), f(b), f(f(a)), ... For simplicity, assuming that the total measures of the following 4 pairs exceeds the threshold: (a,a),(a,b),(b,a),(a,f(b)). An example of search tree for this query is depicted in Figure 4.1. We will follow the execution of the search, firstly until the point where a hypothesis is generated, and then refused by the validation. We will then follow the backtracking steps, and then lastly another forwarding progression until a correct hypothesis is generated and validated.

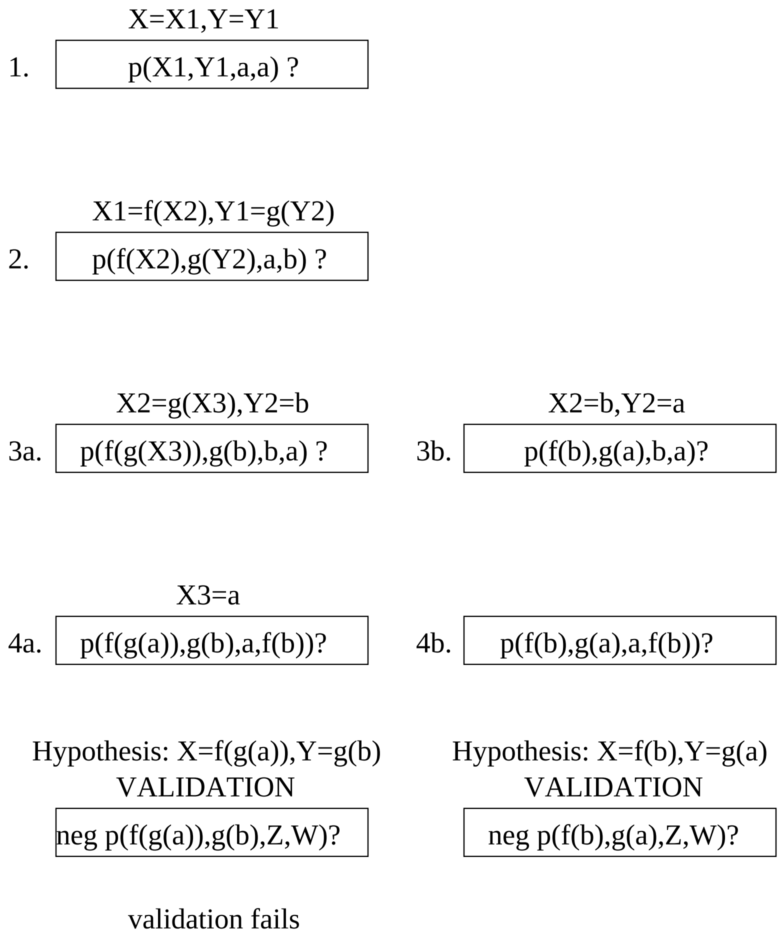

Figure 4.1: An example of search tree for query $\exists \texttt{X Y} \forall \texttt{Z W p(X, Y, Z, W)}$?

4.2.1 Generating candidates

In the example, at the start of the first stage (1.), the variables `(Z,W)` are assigned the values of the first pair `(a,a)`. These variables are subsequently assigned the values of the other pairs in the following stages. The values of the existential variables, on the other hand, become more and more specific as candidates are generated in each stage. At a success node of the Σ_1 search tree in stage 4a. a hypothesis, namely the *current hypothesis* is generated: `X=f(g(a)), Y=g(b)`. This hypothesis will be object to a validation.

4.2.2 Validating hypothesis

The execution now begins to validate the current hypothesis. This involves handling the Π_1 formula $\forall \mathtt{Z},\mathtt{W}\ \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})),\mathtt{g}(\mathtt{b}),\mathtt{Z},\mathtt{W})$. This is done by finding counter example, i.e. carrying out the search for solution of the Σ_1 query $\neg\mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})),\mathtt{g}(\mathtt{b}),\mathtt{Z},\mathtt{W})$. Suppose that this search actually succeeds, i.e. the current hypothesis is refuted. At this point, the Σ_2 search will backtrack within stage 4a (as the Σ_1 search associated with that stage backtracks). Note that the search performed during the validation process can be stopped as soon as one solution is found.

4.2.3 Backtracking

Within stage 4a., the Σ_1 search will backtrack to find other success nodes of the search tree associated with that stage. After completing traversing the search tree associated with stage 4a (without finding any hypothesis), the Σ_2 search backtracks to level 3a. The Σ_2 search continues in stage 3a (as the Σ_1 search associated with that stage continues to find other candidate) from where the variable `X3` was bound to `a` (the only variable in the query at the start this stage `p(f(g(X3)),g(b),b,a)`). Suppose that the search tree associated with stage 3a. does not have any other success node, then the (Σ_2) search backtracks again, to stage 2, to the point where `X2` was bound to `g(X3)` and `Y2` was bound to `b`.

4.2.4 Finding a correct hypothesis

Suppose that the next candidate generated in level 2 is `X2=b, Y2=a`, and the Σ_2 search enters stage 3b. Assuming furthermore that this candidate covers all the other pairs. A validation is carried out in stage 4b., where the system outputs its current hypothesis `X=f(b), Y=g(a)`. If the validation stops and confirms that the hypothesis is correct, the system stops. However, the validation might encounter an infinite loop. In that case, the current output is taken to be the output of the system.

4.3 *RichProlog*

Developed from *iProlog*, *RichProlog* inherits the data structure as well as memory and stack management from *iProlog*. Additional features in *RichProlog* require modification and addition to the existing *iProlog* system. These include adding and modifying data storage, introducing new memory region and using more stack. In this section, each new feature of *RichProlog* will be discussed together with their implications on the structure of the system.

4.3.1 True negation and quantifiers

The introduction of true negation and quantifiers results not only in new predefined keywords, but also in the parsing process of the clauses and queries.

True negation

Negation of a term is achieved by setting a flag in the record representing that term. This is done during the semantic analysis phase as the formulas are parsed. The introduction of true negation also requires some modification in the existing unification procedure. Some other procedures, such as the copying procedure, also need to be modified.

Quantifiers

The queries in level Σ_2 contain both existentially and universally quantified variables. The quantifiers are chosen such that they resemble the symbols used a logic formula: `E` for existential quantifier and `V` for universal quantifier. Apart from these two exceptions, *RichProlog* follows the convention in *iProlog* (e.g. constants start with lower case, while variables start with upper case, etc.). This requires a slight modification in the syntactic analysis phase of the parser.

The presence of quantifiers in the queries also helps to determine the level that queries are in. An existential quantifier followed by a universal quantifier make the system attempt to answer a Σ_2 query. If no quantifiers or just existential quantifier appear in the query, the proof procedure of *iProlog* is carried out. The system also supports queries in level Π_1 (where the proof is done by attempting to find counter example). Queries in levels higher than Σ_2 are simply ignored. Also, it is frequently needed to access all universal variables, or all existential variables which appear in the query. All of these require additional data structure, and extra processing time for each query.

4.3.2 Enumeration of the closed terms

The algorithms described in the previous chapter (section 3.1) assume that an enumeration and measures of the closed terms are available. These should be input by the users. The degree of confirmation should also be modifiable by users. In the implementation of *RichProlog*, these are achieved by predefining some "meta-logic" predicates, as described below.

Enumerations and measures of closed terms

The order of appearance of closed terms in the enumeration, as well as their measures are specified by the predicate

$$\texttt{rank}/3$$

whose format should be `rank(R,T,M)`. `R` identifies the position of the closed term `T` in the enumeration. `M` is the measure of `T`. `rank` has been made a predefined (meta-logic) predicate in *RichProlog*, and is treated differently from a normal logic predicate. The closed terms, together with their measures, as given by the users, are stored in an indexed array. Upon receiving `rank(r,t,m)`, the term `t` and its measure `m` are copied to the position `r` in the array.

Degree of confirmation

The default value for degree of confirmation (or threshold) is 0.5. Its value can be changed by user by using a meta-logic predicate `threshold/1`. This, similar to the predicate `rank`, is treated in a unique way. Upon the input

$$\texttt{threshold}(\alpha).$$

the system modifies the value of a global variable accordingly.

Looping through tuples of closed terms

A query may contain more than one universally quantified variables. Suppose that there are m ($m \geq 1$) universally quantified variables. Each stage in the search procedure starts by assigning a m-tuple of closed terms from the enumeration to these variables. In *RichProlog*, these tuples are enumerated by the sum of their ranks. Thus the tuple whose members are all of rank 0 are assigned to the universal variables in the first stage. In the subsequent m stages, the tuples which contain one closed term of rank 1 and $m-1$ terms of rank 0 are used, etc.

4.3.3 Building program

The building program process (see the algorithm to build a program, Figure 3.3) requires a data structure to store the set Γ. If the head of some rule associated with a clause from $\mathcal{S}$ unifies with a member of Γ, then the rule is added to form the logic program. In that case, the terms in its body are also added to the set Γ. The set Γ has been implemented as a FIFO queue. Its initial elements are the terms from the query. A term will be added to the set Γ at most once. This is guaranteed by setting a special flag in the representation of the term. By ensuring this, each rule will be added into the program at most once.

To add a rule into the program, we construct the a record for the rule from the corresponding clause in $\mathcal{S}$. Similar to *iProlog*, the rules with the same functor in the head are grouped into a linked list which can be accessed from the record of the functor. Thus each record for a functor contains the pointer to 2 linked lists for the rules from 2 programs T_1 and T_2.

When a term is popped from the queue for Γ, a unification is performed to test if it unifies with the heads of the rules. Potentially, the negation of each of the disjuncts of a clause can be the head of a rule associated with that clause. Thus the terms which appear in the clauses of the set $\mathcal{S}$ are grouped together according to their functor.

Another matter involved in the building program process is the storage of a program. The storage for these rules should be allocated in such a way that after the proof procedure finishes, it can be recovered quickly. This is the behavior of copy stack or heap as described in the previous section (part 4.1.2).

4.3.4 Searching and backtracking

Backtracking between stages in a Σ_2 search requires an additional stack to store the state of the execution in a stage before it advances into the next one. This includes the bindings of variables which appear in the Σ_1 queries of the stage. It also includes indices of the m-tuples of closed terms which are used to generate the candidate at the stage. Apart from these, there are pointers to the rules and goals of the Σ_1 search which also need to be restored upon backtracking between stages. The total measure of the m-tuples of closed terms at a stage is also stored in this stack.

Another issue is that the variables which appear in a candidate may be bound to some values during the Σ_1 search for subsequent candidates. Its binding must be stored so that they can be recovered upon backtracking between stages. It is therefore necessary to copy an image of the candidate from a stage. The copy is used as a starting point for the search associated with the next stage.

The Σ_2 search only backtracks to its previous stage if the Σ_1 search tree associated with its current stage reaches its last branch. It is necessary to detect when this happens. One flag in an environment record is sufficient for this purpose. It will indicate whether the current node in the search tree is in the last branch of the search tree or not. Thus the flag for a node in the search tree will be set to indicate that it is in the last branch if and only if that node is the last child node of its parent, and its parent is also in the last branch of the search tree.

4.3.5 Primary evaluations

The search performed in *RichProlog* can be regarded as consisting of a number of interactions of the search procedure in a Prolog system. Implementation of *RichProlog* therefore involves issues which are concerned in any Prolog system, but in larger scale. For example, employing different strategies for the unification procedure may have quantitatively significant impact on the performance of the system. On the other hand, there are unique features of *RichProlog* that are not present in an ordinary Prolog system. This part discusses on some features of the current implementation of *RichProlog*.

Evolving from *iProlog*, *RichProlog* inevitably inherits its characteristics. This is reflected mainly in the unification algorithm, with the use of structure copying, which gives fast access, but at the cost of constructing terms. It therefore also requires more space for the variable bindings. Structure sharing, on the other hand, is known for its economical management of space, and does not require time for constructing instances of terms. However, accessing variable values are slowed by successive references. The choice between different unification algorithms really depends on the particular applications.

Note that unification is also carried in the building program process, when literals in clauses are unified with member of the set Γ (see section 3.1.5). Here the task is to check if two terms unify, the result of unification is not used later. It can be seen that in this case, structure sharing is more preferable.

In *RichProlog* copying of terms is also performed when a candidate generated at one stage is passed to the next stage. Here the candidates can be considered as being "hardwired" to the parse tree of the query formula. The values of the candidates are used in all subsequent stages. A sequence of references as it would be if one followed the methodology of structure sharing is therefore not preferred.

4.3.6 Possible improvements

This part will discuss some possible enhancement to the current *RichProlog* system. As *RichProlog* is a reflection of *iProlog* in a larger scale, improvement in *iProlog* would definitely have deep effects on *RichProlog*. The unique features of *RichProlog* can also be exploited for better performance. The system can be improved in its various aspects, for example there can be improvement in the way backtracking is implemented as well as some other optimization such as last-call optimization, clause indexing. A nontrivial suggestion which may draw attention is the implementation of a parallel interpreter for *RichProlog*.

Improvement in backtracking

Firstly, *intelligent backtracking* ([Bru81, Con87]) can be implemented. In this method, bindings of the variables are analysed, and the variable bindings which cause the failure of unification are identified. The search can then backtrack to the point where these bindings were performed. This enhancement can be of more importance when applied to *RichProlog*. Not only can the search backtrack within the (Σ_1) search tree of a stage, but it can also help to quickly backtrack between stages.

To further improve backtracking, one could also consider the validation process. Note that a current hypothesis is refuted not because of failure in unification, but because of the success in the validation process. The analysis can be more complicated, but the advantages may be encouraging. As an example, consider the Σ_2 query

$$\exists \mathtt{X},\mathtt{Y} \forall \mathtt{Z} \mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z})?$$

Suppose that in stage 1 the candidate $(\mathtt{X1},\mathtt{Y1}) = (\mathtt{f(X2)},\mathtt{g}(\mathtt{a},\mathtt{b}))$ is generated. Then in the last stage a hypothesis $(\mathtt{X},\mathtt{Y}) = (\mathtt{f}(\mathtt{g}(\mathtt{f}(\mathtt{a}),\mathtt{b})),\mathtt{g}(\mathtt{a},\mathtt{b}))$ is generated. If in the validation process, it is discovered that the value `g(a,b)` of `Y1` is the cause of the success (e.g. $\neg\mathtt{q}(\mathtt{W1},\mathtt{g}(\mathtt{a},\mathtt{b}),\mathtt{W2})$ is in the program T_2), then the Σ_2 search can backtrack directly to stage 1 where `Y1` was bound to `g(a,b)`.

Clause indexing

Clause indexing ([War77, War83, Boi93]) can also be implemented. This, in its original form, is an optimization in which clauses are grouped according to their head. The discussion so far assumes that the program is stored as a number of sets of clauses where clauses in a set have the same functor in their head. As an optimization, these sets can be further divided by some other criteria, such as the number of arguments in the head, or the type of the first argument of the head. This can be implemented in *RichProlog* in exactly the same way. The name, however, should be *rule indexing*, since we would be grouping rules, rather than clauses. Note that in *RichProlog* a trivial criteria for grouping the rules can be the sign of the literal in their head.

Last-call optimization

Another improvement can come from implementing last-call optimization ([Bru80, Boi93, KS78]). This optimization helps to save space which otherwise would be needed in the stack when the search is at the last child of an *and* node, which is itself the last child of an *or* node in the search tree. Similar to the methods mentioned above, this optimization would have its effect more realizable in *RichProlog* than in a Prolog interpreter.

Parallel *RichProlog* interpreter

The system can also be enhanced with parallelism. Given the underlying architecture's support for multiprocessing, the system can be implemented so that answering a query can be carried out in parallel. This part will discuss some aspects of the inherent parallelism in implementation of *RichProlog*. Since each stage in the Σ_2 search is equivalent to a Σ_1 search, parallelism can be implemented in each stage in exactly the same way as they have been implemented in parallel Prolog interpreters. *RichProlog*, however, has its own characteristics which are subject of parallelism. In the following parts, we will firstly summarize some of the existing concepts that have been introduced in parallel logic programming ([Kac90, Con87, Qui88]). Then we will discuss how parallelism can be exploited in an implementation of *RichProlog*.

In the implementation of a Prolog system, parallelism can be present at different levels, as classified in [Kac90]:

1. Parallelism within unification
2. Parallelism among unification
3. Parallelism in data processing
4. Parallelism in the control strategy

In implementing *RichProlog*, parallelism can be present in a higher level:

5. Parallelism among stages

Basically, parallelism within unification is applicable in the case where the literals in the heads of the rules have more than one argument, and therefore the unification of each argument can be performed in parallel. Care must be taken when the arguments have common variables. Either parallelism needs to be avoided, or the results of unification need to be checked for consistency ([ea85]). On the other hand, parallelism among unification is the case when different unification processes between the goal and the heads of different rules are performed in parallel. Parallelism in data processing (also called *stream parallelism*, [CK81]) is the case when there are common variables in the goals. These goals can be solved in pipeline. The partial results of unification on these variables produced by a goal can be immediately consumed by other goals. At the control level, different branches of the search tree can be explored in parallel by different processes. Thus there are *and-parallelism* and *or-parallelism*, depending on the kind of nodes in the search tree where a process can be branched.

It can be seen in the Σ_2 search tree that the relationship between stages has a so-called "producer-consumer" characteristic. The candidates generated in a stage is used in the next stage. The idea of parallelism among stages is therefore similar to that of stream parallelism. The partial value of a candidate can immediately be made available to the Σ_1 search of the next stage. This should be compared with how stream parallelism is implemented. Note that while stream parallelism can be implemented in such a way that make the logic programming language either concurrent or parallel ([Con87]), our aim here is to hide parallelism from the application level. This can be achieved, with a notice that the *data flow ordering* and *data dependency analysis* in this case are trivial tasks.

4.4 An application

In this last section, an example will be given to show an application of *RichProlog*. It will be used to implement a learner in the following learning problem.

4.4.1 The problem

In this problem, strings are nonempty sequences of `a, b`, while patterns are nonempty sequences composed of `a,b` or the variables $\mathtt{V1}, \mathtt{V2}, \ldots$. Suppose that $\mathtt{p} = \mathtt{p}_1 \ldots \mathtt{p}_\mathtt{m}$ is a pattern, where $p_1, \ldots, p_m$ belong to $\{\mathtt{a}, \mathtt{b}, \mathtt{V1}, \mathtt{V2}, \ldots\}$. An *instance* of `p` is a string obtained by replacing all the variables in `p` by strings, with the same replacement for identical occurrence of variables. A string `s` and a pattern `p` are said to *match* if the `s` is an instance of `p`. We say that the pattern $\mathtt{p} = \mathtt{p}_1 \ldots \mathtt{p}_\mathtt{m}$ where $\mathtt{p}_1, \ldots, \mathtt{p}_\mathtt{m} \in \{\mathtt{a}, \mathtt{b}, \mathtt{X}_1, \mathtt{X}_2, \ldots\}$ and the string $\mathtt{s} = \mathtt{s}_1 \ldots \mathtt{s}_\mathtt{n}$ where $\mathtt{s}_1, \ldots, \mathtt{s}_\mathtt{n} \in \{\mathtt{a}, \mathtt{b}\}$ are of length m and n respectively. Given a set of instances of a pattern, the problem is to identify the pattern.

Following the paradigm given in [OSW86], this identification can be formalized as follows. The hypotheses are possible patterns. The environment is the exhibition of instances that are given to the learner. The learner should have the following behaviors. Given a subset of the environment (a set of data), it is expected to return its hypothesis, which belongs to the hypothesis space. The identification has a limiting nature, i.e. given a finite set of data, the learner is not expected to necessarily return a correct hypothesis. However, as more and more data are given, conjectures made by the learner should converge to the correct one.

On the other hand, identifying a pattern which should agree with a set of strings is indeed answering a Σ_2 query *"does there exist a pattern which matches all of the given strings ?"*. This equivalence of formalizing the problem suggests an application of *RichProlog*. The next part will investigate how it can be solved using *RichProlog*.

4.4.2 Specifying the problem in *RichProlog*

In this part, we will discuss different aspects of the problem. These include the representation of the hypothesis space, the representation of the environment and lastly, the learner. It will be seen that these are best handled in the *RichProlog*'s partial mode, where the application level has the flexibility to optimize and also avoid the problems discussed in 3.4. In the following paragraphs, *data* means an arbitrary string, *positive data* refers to the instances of the pattern to be learned and *negative data* refers to other strings.

The hypothesis space

The learner should have access to its hypothesis space (the set of all patterns). One way to do this is to provide it with the capability to generate the patterns. One would start by generating all patterns of length 1, then 2, etc. Thus one needs the generation of all positive integers. This is done by the following two rules:

```
numb(z).
numb(s(N)) :- numb(N).
```

As discussed earlier (section 3.4.3), if one associates the rule `neg numb(N) :- neg numb(s(N))` to the second rule, then the proof procedure will never get out of an infinite loop in a call to `neg numb(X)`. The rules introduced above are only for the purpose of generating natural numbers. Thus the rules `neg numb(N) :- neg numb(s(N))` can be considered as undesirable.

The learner generates hypotheses (patterns) of increasing length. The rules for this can be examined in Figure 4.2. Basically, the predicate

```
pattern(Pi, N)
```

is true for by pattern `Pi` of length `N`. Note that these rules are only used in generating patterns. Thus they are used in the logic program T_1 only. The validation process does not require this generation, and therefore they are not included in T_2. This optimization is achieved only when *RichProlog* is working in its partial mode.

The environment

The environment is presented to the learner by using the predicate

`p/1`

Thus before asking a query, users should insert a number of statement specifying the (positive) data given to the learner.

The learner can be thought of as having access to all possible strings. At any point, it may consider all strings which are not yet given to it as not being instance of the pattern to be learned. Obviously, a number of these strings will be presented to it at some later point. It will then have to change its hypothesis accordingly. Similar to the discussion about the hypothesis, the learner can be provided with a mechanism to generate all the data, and to pick a number of the strings that are not given to it as negative data. The approach chosen in this application is to list the strings for the learner. Note that this is different from providing both negative and positive data to the learner. The list of strings could instead be generated by the learner itself.

The learner

From a set of instances, the learner must be able to generate and validate its hypothesis. The basic operation needed is that given a pattern and a string, it must be able to determine whether or not they match. The following predicate takes as input a pattern and a string, it is true when they match:

`match/2`

One strategy for the learner is to examine the shortest pattern which matches all the positive data, and does not match the others. Since the patterns are generated in order of their length, this strategy is implicitly employed by the learner in this problem. Also, it can be easily verified that a pattern of length `n` will not match at least one of the negative data of length `n`, if there is any such string. Due to the limiting nature of the problem, if a pattern `Pi` of length n is to be learned, then eventually all positive data of length n will be given to the learner. At that point, its assertion about the negative data of length `n` will be correct. The learner, therefore needs to validate its hypothesis using only strings of length n. Implementation of `match` can be seen in Figure 4.2. Essentially `match` only succeeds when the pattern and the strings are of the same length and match.

We will now look at the query that can be handled by the Σ_2 proof procedure. The learner needs to hypothesize a pattern which matches all the positive data, and does not match any string considered to be a negative data. It does so by first generating possible hypotheses, and then validating them. Generating a hypothesis requires the query:

$$\exists \texttt{Pi},\texttt{N}[\texttt{pattern}(\texttt{Pi},\texttt{N}) \wedge \forall \texttt{W}$$
$$((\texttt{length}(\texttt{W},\texttt{N}) \wedge \texttt{match}(\texttt{Pi},\texttt{W}) \rightarrow \texttt{p}(\texttt{W})) \wedge (\texttt{length}(\texttt{W},\texttt{N}) \wedge \texttt{mismatch}(\texttt{Pi},\texttt{W}) \rightarrow \neg\texttt{p}(\texttt{W})))]$$

Validating the hypothesis involves the following refuting sentence, which aims at finding counter-examples for the candidate:

$$\exists \texttt{W}[\neg\texttt{pattern}(\texttt{Pi},\texttt{N})\vee$$
$$(\texttt{length}(\texttt{W},\texttt{N}) \wedge \texttt{match}(\texttt{Pi},\texttt{W}) \wedge \neg\texttt{p}(\texttt{W})) \vee (\texttt{length}(\texttt{W},\texttt{N}) \wedge \texttt{mismatch}(\texttt{Pi},\texttt{W}) \wedge \texttt{p}(\texttt{W}))]$$

Here `mismatch` is used in place of ¬`match`. This is because leaving ¬`match` would cause *RichProlog*'s refusal to the queries. The predicate `p`, on the other hand, does not appear in the programs, and thus its negation in the query does not cause any problem.

Transforming the first query results in the following rules (presented in *RichProlog* syntax)

```
 q(Pi,N,W)  :-  pattern(Pi,N),q1(Pi,N,W).
q1(Pi,N,W)  :-  p(W),q2(Pi,N,W).
q1(Pi,N,W)  :-  neg p(W),q3(Pi,N,W).
q2(Pi,N,W)  :-  length(W,N1),distinct(N1,N).
q2(Pi,N,W)  :-  match(Pi,W).
q3(Pi,N,W)  :-  length(W,N1),distinct(N1,N).
q3(Pi,N,W)  :-  mismatch(Pi,W).
```

Note that in the validation process, the pair `Pi,N)` is assigned value of a candidate, thus ¬`pattern(Pi,N)` will always fail. Knowing this, it can be removed from the validation query. The rule introduced in the transformation for negation of the query are:

```
neg q(Pi,N,W)  :-  p(W),length(W,N),mismatch(Pi,W).
neg q(Pi,N,W)  :-  neg p(W),length(W,N),match(Pi,W).
```

The two logic programs are given in Figure 4.2. Note that the two programs share most of the rules. It is therefore reasonable to put them in one place. Each of them will be added with a separate set of rules presented above. Note that the program T_2 (program for the validation) does not contain rules to generate patterns.

It can be seen that the programs are not optimal. They can be optimized in many ways. For example, the length of the pattern could be effectively determined by the length of the shortest positive data string. The programs presented here are only for the purpose of illustrating the applicability of the system.

Rules in T_1 only:

1. numb(z).
2. numb(s(N)) :- numb(N).
3. distinct(s(N),z).
4. distinct(z,s(N)).
5. distinct(s(N1),s(N2)) :- distinct(N1,N2).
6. select(N,s(N)).
7. select(N1,s(N)) :- select(N1,N).
8. pattern(Pi,s(N)) :- numb(N), pattern1(Pi,s(N)).
9. pattern1(gen pattern,z).
10. pattern1(Pi,s(N1)) :- pattern1(Pi,N1).
11. pattern1(subst_a(Pi,s(N1)),s(N1)) :- pattern1(Pi,N1).
12. pattern1(subst_b(Pi,s(N1)),s(N1)) :- pattern1(Pi,N1).
13. pattern1(equal var(Pi,s(N2),s(N1)),s(N1)) :- select(N2,N1),pattern1(Pi,N1).

Common rules in T_1 and T_2:

14. length(e,z).
15. length(a(W),s(N)) :- length(W,N).
16. length(b(W),s(N)) :- length(W,N).
17. symb_a(a(W),s(N)) :- length(W,N).
18. symb_a(a(W),N) :- symb_a(W,N).
19. symb_a(b(W),N) :- symb_a(W,N).
20. symb_b(b(W),s(N)) :- length(W,N).
21. symb_b(a(W),N) :- symb_b(W,N).
22. symb_b(b(W),N) :- symb_b(W,N).
23. match(gen pattern,W).
24. match(subst_a(Pi,N),W) :- symb_a(W,N), match(Pi,W).
25. match(subst_b(Pi,N),W) :- symb_b(W,N), match(Pi,W).
26. match(equal var(Pi,N2,N1),W) :- symb_a(W,N1), symb_a(W,N2),match(Pi,W).
27. match(equal var(Pi,N2,N1),W) :- symb_b(W,N1), symb_b(W,N2),match(Pi,W).
28. mismatch(subst_a(Pi,N),W) :- symb_b(W,N).
29. mismatch(subst_b(Pi,N),W) :- symb_a(W,N).
30. mismatch(equal var(Pi,N2,N1),W) :- symb_a(W,N1), symb_b(W,N2).
31. mismatch(equal var(Pi,N2,N1),W) :- symb_b(W,N1), symb_a(W,N2).
32. mismatch(subst_a(Pi,N),W) :- mismatch(Pi,W).
33. mismatch(subst_b(Pi,N),W) :- mismatch(Pi,W).
34. mismatch(equal var(Pi,N1,N2),W) :- mismatch(Pi,W).

Figure 4.2: Logic programs T_1 and T_2 for the pattern application

The programs have been tested with a number of different patterns, including `V0V0V0`, `V1aV1a` and `V3abV0`. We will illustrate the limit behavior of the learner with the pattern `V0V0V0`. Firstly, when we present to the learner the positive data `aaa` and we assert, on behalf of the learner, that the following are presumably negative data: `a, b, aa, ab, ba, bb, bab, bbb, aba` then the learner outputs the hypothesis `V2aa`. Note that if `baa` is also in the list of presumably negative data, then the hypothesis output by the learner is `aaV0`. We are now informed that `bbb` is actually a positive data. Thus we retract it from the list of presumably negative data, and assert that it is a positive data. Then the learner outputs the hypothesis `V1V1V0`. Since it has not seen `bba` in the list of positive data, the learner can assume that `bba` is a negative data. Now it will output the correct hypothesis `VOVOVO`. Since `bba` is actually a negative data, it will never be retracted from the list of presumably negative data. And in fact, none of the strings in this list is a positive data, thus the learner will keep this hypothesis.

To complete the discussion on this application, there are a few remarks to be made about the user interaction that has been supported. *RichProlog*'s I/O is inherited from *iProlog*. It is able to read inputs from a file and write outputs to another file. These are exploited for reusable testing cases. Also, the functional representation of the strings and patterns are not very comprehensible to users. Thus a number of Perl scripts have been written to translate users readable inputs into Prolog terms, and translate the results into readable format.

Chapter 5

Conclusion and future work

This thesis has introduced the notion of generalized logical consequence, which formalizes the notion of logical consequence in other frameworks, such as classical logic, ILP or nonmonotonic reasoning. In these frameworks, the intended logical consequences of a theory are identified by isolating the intended models. This is often done firstly by assuming that the intended models are of some kind e.g., Herbrand structures. Secondly, a minimality condition is imposed on the models, either explicitly or implicitly, usually by extending the theories. Note that these are not always the prerequisites of the frameworks, but can be present in some characterizing results of other settings, such as requirement on the hypotheses space. By formalizing the notion of a class of possible worlds and the notion of minimality (which is based on the notion of a set of possible observations), we achieve the generalized notion of logical consequence.

We have shown (in chapter 2 and in the appendix) a number of properties of this new notion of logical consequence. Some properties of the classical notion of logical consequence, such as the idempotence property, are preserved. However, many others are not. Especially, it has been shown that theories which have the same sets of generalized logical consequences do not necessarily have the same sets of minimal models. This is one of many arguments for showing that the structure of the set of generalized logical consequences of a theory is very complex. The results also show that for some choices of $\mathcal{O}$, for example, when $\mathcal{O}$ is closed under negation, the generalized notion of logical consequence is simplified. It can also be seen that a large number of results frequently involve some particular choices of $\mathcal{W}$ and $\mathcal{O}$, such as $\mathcal{W}$ being the class of Herbrand structures, $\mathcal{O}$ being some subset of basic sentences or atomic sentences. This is because these settings are quite common in the existing frameworks. However, many other settings can also be of interest. For example, one may wish to investigate some subsets of the set of Herbrand structures. Note that not every of these subsets can be identified using only formulas in first order logic, but they can always be captured by $\mathcal{W}$. The general logical framework, in that case, will prove to be the appropriate choice.

The structure of the set of generalized logical consequences of a theory can be best studied with reference to other theories in the framework. Specifying the particular set of possible theories makes it feasible to characterize the generalized logical consequences of theories in the set. In particular, they structure the generalized logical consequences of a theory in a hierarchy over it. In this hierarchy, a higher level is obtained from the lower ones by either compactness and weak compactness properties. However, this also suggests that different choices of the set of possible theories will structure the logical consequences of the theories differently. With some set the hierarchies can be complete, in the sense that they contain all consequences of a theory, while with others they are not. One open problem is to investigate the characteristics of the sets of possible theories which provide the complete hierarchies over its members.

With a particular setting of the framework (e.g. the possible worlds are Herbrand structures, possible observations are atomic sentences and possible theories are sets of Horn clauses) we obtain the working context of Prolog. Prolog, with its deductive inference ability, can then be regarded as providing a Σ_1 proof procedure in this restrictive setting of the framework. Being declarative, Prolog allows users to specify a (deductive) problem and then it will work out how to solve it. However, many problems arising in AI are concerned with induction rather than deduction. Prolog is then used in the implementation of the ILP systems. These systems, however employ the deductive inference mechanism of Prolog as their core driver. In order to use the deductive power of Prolog, they need to implement the inference rules and operators such as the refinement operator ([Sha83]), inverting entailment ([Mug95, Mug98]). Programmers using these systems still need to be concerned with issues such as mode and type declaration ([Mug01]).

RichProlog can be regarded as a natural extension of Prolog. It assumes the same setting on the framework as the setting for Prolog. Most importantly, it can perform both deduction and induction, while maintaining the declarative property. The declarativeness should be the ultimate advantage of *RichProlog* over other ILP systems. Users can just describe the problem, and the induction as well as deduction are worked out automatically. This maintains the equation

$$\textbf{Algorithm} = \textbf{Logic} + \textbf{Control} \text{ (Kowalski, [Kow79])}$$

Moreover, within this setting, it can be shown that the formulas on the Σ_2 level correspond to the concepts that are learnable in the limit in learning paradigm. This has been shown in the application (4.4) of learning a nonerasing pattern languages from positive data only.

The implementation of the system still has room for improvements, as discussed in section 4.3.6. In particular, one may attempt various optimizations such as clause indexing, intelligent backtracking, etc. The inherent parallelism problem of implementing *RichProlog* also promises a major improvement. Analysis of different unification algorithms (such as structure copying vs. structure sharing) would be valuable. Applications of the system include that of any ILP system, such as drug design and protein primary-secondary shape prediction. Furthermore, it is always beneficial to customize the system to a particular application.

Bibliography

[Boi93] Patrice Boizumault. *The implementation of Prolog.* Princeton University Press, 1993.

[Bru80] M. Bruynooghe. The memory management of prolog implementation. *Logic programming workshop '80*, pages 12–20, 1980.

[Bru81] M. Bruynooghe. Intelligent backtracking for an interpreter of horn clause logic programs. In B. Dömölki and T. Gergely, editors, *Mathematical Logic in Computer Science*, pages 215–258. North-Holland Publ., 1981.

[CK81] J.S. Conery and D.F. Kibler. Parallel interpretation of logic programs. pages 163–170. ACM, 1981.

[Con87] John S. Conery. *Parallel execution of logic programs.* Kluwer Academic Publishers, 1987.

[Doe94] Kees Doets. *From logic to logic programming.* The MIT Press, 1994.

[ea85] N. Ito et al. Data-flow based execution mechanism of parallel and concurrent prolog. *New Generation Computing*, (2):15–41, 1985.

[EFT89] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical logic.* Spinger-Verlag, 1989.

[Ham78] A. G. Hamilton. *Logic for mathematicians.* Cambridge University Press, 1978.

[Kac90] Péter Kacsuk. *Execution models of Prolog for Parallel Computers.* The MIT Press, 1990.

[Kow79] R. Kowalski. Algorithm = logic + control. *Comm. ACM*, (22):424–436, 1979.

[KS78] Feliks Kluźniak and Stanislaw Szpakowicz. *Logic for mathematicians.* Cambridge University Press, 1978.

[Luk90] Witold Lukaszewicz. *Non-monotonic reasoning.* Ellis Horwood, 1990.

[MR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, (19,20):629–679, 1994.

[MSS] Eric Martin, Arun Sharma, and Frank Stephan. *Deductive and inductive Logic*. To be published.

[MSS01] E. Martin, A. Sharma, and F. Stephan. A general theory of deduction, induction and learning. *Fourth International Conference on Discovery Science*, 2001.

[Mug95] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13: 245–286, 1995. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/InvEnt.ps.gz`.

[Mug98] S. Muggleton. Completing inverse entailment. In C.D. Page, editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 245–249. Springer-Verlag, Berlin, 1998. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/ie.ps.gz`.

[Mug01] S. Muggleton. CProgol4.4: a tutorial introduction. In *Inductive Logic Programming and Knowledge Discovery in Databases*. Springer-Verlag, 2001. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/progtuttheo.ps.gz`. To appear.

[OSW86] Daniel N. Osherson, Michael Stob, and Scott Weinstein. *Systems that learns*. The MIT Press, 1986.

[Qui88] Michael J. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, 1988.

[Sha83] E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

[Sri98] Sashi Mohan Srivastava. *A course in Borel sets*. Springer, 1998.

[SS83a] C.A. Sammut and R.A. Sammut. The implementation of unsw-prolog. *Australian Computer Journal*, 15(2):42–51, 1983.

[SS83b] C.A. Sammut and R.A. Sammut. Prolog: A tutorial introduction. *Australian Computer Journal*, 15(2):42–51, 1983.

[SS97] Leon Sterling and Ehud Shapiro. *The art of Prolog*. The MIT Press, 1997.

[War77] D.H. Warren. Implementing prolog: Compiling predicate logic programs. Technical report, D.A.I. Edinburgh, 1977.

[War83] D.H. Warren. An abstract prolog instruction set. Technical report, SRI International, 1983.

Appendix A

Generalized Logical Consequences and the Closed World Assumption