USING MODEL GENERATION THEOREM PROVERS FOR THE COMPUTATION OF ANSWER SETS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ORKUNT SABUNCU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JULY 2009

Approval of the thesis:

USING MODEL GENERATION THEOREM PROVERS FOR THE COMPUTATION OF ANSWER SETS

submitted by **ORKUNT SABUNCU** in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Engineering Department, Middle East Technical University by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Müslim Bozyiğit _____
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Ferda Nur Alpaslan _____
Supervisor, **Computer Engineering**

Examining Committee Members:

Prof. Dr. Varol Akman
Computer Engineering Dept., Bilkent University _____

Assoc. Prof. Dr. Ferda Nur Alpaslan
Computer Engineering Dept., METU _____

Prof. Dr. Mehmet Reşit Tolun
Computer Engineering Dept., Çankaya University _____

Assoc. Prof. Dr. Nihan Kesim Çiçekli
Computer Engineering Dept., METU _____

Assist. Prof. Dr. Pınar Şenkul
Computer Engineering Dept., METU _____

Date: **2/7/2009**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ORKUNT SABUNCU

Signature :

ABSTRACT

USING MODEL GENERATION THEOREM PROVERS FOR THE COMPUTATION OF ANSWER SETS

Sabuncu, Orkunt

Ph.D., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Ferda Nur Alpaslan

July 2009, 98 pages

Answer set programming (ASP) is a declarative approach to solving search problems. Logic programming constitutes the foundation of ASP. ASP is not a proof-theoretical approach where you get solutions by answer substitutions. Instead, the problem is represented by a logic program in such a way that models of the program according to the answer set semantics correspond to solutions of the problem.

Answer set solvers (Smodels, Cmodels, Clasp, and Dlv) are used for finding answer sets of a given program. Although users can write programs with variables for convenience, current answer set solvers work on ground logic programs where there are no variables. The grounding step of ASP generates a propositional instance of a logic program with variables. It may generate a huge propositional instance and make the search process of answer set solvers more difficult.

Model generation theorem provers (Paradox, Darwin, and FM-Darwin) have the capability of producing a model when the first-order input theory is satisfiable. This work proposes the use of model generation theorem provers as computational engines for ASP. The main motivation is to eliminate the grounding step of ASP completely or to perform it more intelligently

using the model generation system. Additionally, regardless of grounding, model generation systems may display better performance than the current solvers. The proposed method can be seen as lifting SAT-based ASP, where SAT solvers are used to compute answer sets, to the first-order level for tight programs.

A completion procedure which transforms a logic program to formulas of first-order logic is utilized. Besides completion, other transformations which are necessary for forming a first-order theory suitable for model generation theorem provers are investigated. A system called Completor is implemented for handling all the necessary transformations. The empirical results demonstrate that the use of Completor and the theorem provers together can be an effective way of computing answer sets. Especially, the run time results of Paradox in the experiments has showed that using Completor and Paradox together is favorable compared to answer set solvers. This advantage has been more clearly observed for programs with large propositional instances, since grounding can be a bottleneck for such programs.

Keywords: Answer set programming, Model generation theorem prover, Grounding, Answer set semantics, Completion semantics

ÖZ

YANIT KÜMELERİNİN HESAPLANMASINDA MODEL OLUŞTURABİLEN TEOREM İSPATLAYICILARININ KULLANILMASI

Sabuncu, Orkunt
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ferda Nur Alpaslan

Temmuz 2009, 98 sayfa

Yanıt kümesi programlama, arama problemlerinin çözülmesinde kullanılan bildirimsel bir yaklaşımdır. Sonuçların yerine koyma yöntemiyle alındığı ispat tabanlı bir yaklaşım değildir. Problemi temsil eden mantık program öyle oluşturulur ki programın yanıt kümesi anlam bilimine göre bulunan modelleri, problemin çözümlerine karşılık gelir.

Mantık programının yanıt kümeleri yanıt kümesi çözümleyicisi (Smodels, Cmodels, Clasp ve Dlv) tarafından bulunur. Kullanıcılar kullanım rahatlığı açısından değişkenlerin olduğu programlar yazsalar da, şimdiki yanıt kümesi çözümleyiciler değişkenlerin olmadığı yalın programlarla çalışmaktadır. Yanıt kümesi programlamanın indirgeme süreci, değişkenlerin olduğu bir programın sadece önermelerden oluşan halini oluşturur. İndirgeme süreci çok büyük bir program oluşturabilir. Bu da yanıt kümesi çözümleyicilerinin işini zorlaştırabilir.

Model oluşturabilen teorem ispatlayıcılar (Paradox, Darwin ve FM-Darwin) niceleme mantığında verilen bir teori doğru olduğu zaman, teoriyi doğrulayan modeli oluşturabilme yeteneğine sahiptir. Bu çalışmada model oluşturabilen teorem ispatlayıcılarının yanıt kümesi programlamada hesaplayıcı sistem olarak kullanılması önerilmektedir. Ana motivasyon yanıt kümesi programlamada bulunan indirgeme sürecinin kaldırılmasını ya da model oluşturabilen sis-

temler tarafından daha verimli bir şekilde yapılmasını sağlamaktır. Ayrıca, indirgeme süreci dışında, model oluşturabilen teorem ispatlayıcılar şimdiki yanıt kümesi çözümleyicilerinden daha iyi performans gösterebilir. Önerilen yöntem yanıt kümelerini doğruluk çözümleyicilerini kullanarak bulmaya çalışan yöntemin niceleme mantığı seviyesine çıkarılması olarak görülebilir.

Bu çalışmada bir mantık programını niceleme mantığı önermelerine çeviren tamamlama işleminden yararlanılmıştır. Tamamlama dışında, model oluşturabilen teorem ispatlayıcılara uygun girdiyi oluşturmak için gerekli dönüşümler de incelenmiştir. Tüm bu dönüşümleri gerçekleştirebilen Completor adında bir sistem geliştirilmiştir. Deneysel sonuçlar Completor ve teorem ispatlayıcılarını birlikte kullanmanın yanıt kümelerinin hesaplanmasında etkili bir yöntem olduğunu göstermiştir. Özellikle Paradox'un deneylerdeki zaman sonuçları Completor ve Paradox'u birlikte yanıt kümesi çözümleyiciler yerine kullanmanın daha avantajlı olacağını göstermiştir. Bu avantaj sadece önermelerden oluşan halinin çok büyük olduğu programlar için daha açık olarak gözlenmiştir; çünkü bu tür programlar için indirgeme süreci darboğaz teşkil etmektedir.

Anahtar Kelimeler: Yanıt kümesi programlama, Model oluşturabilen teorem ispatlayıcılar, İndirgeme, Yanıt kümesi anlam bilimi, Tamamlama anlam bilimi

To my family

ACKNOWLEDGMENTS

I must thank my parents, Emine and İbrahim Sabuncu for their never-ending support and encouragement. Although they were far from Ankara, I always felt their support by my side. I must also thank my brother, Ali Sabuncu. Besides his encouragement, he was a wonderful home mate, which made life more enjoyable for me throughout this work.

I would like to thank my advisor, Ferda Nur Alpaslan for her supervision and guidance throughout this work. Her guidance improved my research skills and helped me complete this thesis.

I am grateful to my advisory committe members Varol Akman and Nihan Kesim Çiçekli for their comments and suggestions.

I would like to thank Esra Erdem for her suggestions. From start to end of this work, I am indebted to her for valuable comments and suggestions. She also helped me meet many great people from the research community.

I am grateful to Vladimir Lifschitz, Ilkka Niemelä, and Tomi Janhunen for useful comments and discussions.

I also thank my friends for providing support and friendship that I needed.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ASP Answer Set Programming

DIG Disjunctions of Implicit Generalizations

DPLL Davis-Putnam-Logemann-Loveland Procedure

EPR Effectively Propositional Logic

ME Model Evolution

QG Quasigroup Existence

SAT Satisfiability Checking

SLD Selective Linear Definite Resolution

SLDNF Selective Linear Definite with Negation as Failure Resolution

TPTP Thousands of Problems for Theorem Provers

CHAPTER 1

INTRODUCTION

Using computers to solve problems which are impractical for humans to do, is one of the main subjects of computer science. We have to first formulate the problem so that a computer can solve it. This can be achieved by instructing a computer the method of solving the problem step-by-step. Furthermore, many studies of AI investigate more intelligent ways of describing and solving a problem. In *declarative programming*, one describes what the problem is without dealing with the details of how it can be solved.

Answer set programming (ASP) is a declarative approach for solving search problems. A search problem is represented by a logic program whose models, according to answer set semantics, correspond to solutions of the problem. In this sense, it departs from logic programming with Prolog since it is not a proof-theoretical approach where you get solutions as answer substitutions in proofs. ASP originated from logic programming and nonmonotonic reasoning during the late 1990s [35, 41, 39]. With the emergence of fast answer set solvers (Smodels[1], Dlv[2], Cmodels[3], Assat[4], Clasp[5]) and because of its highly declarative nature, ASP has become an effective and widely accepted paradigm for knowledge representation and reasoning tasks [1]. Among its areas of application, we can list planning [35], configuration [50], and diagnosis [6].

Refutational theorem provers search for the unsatisfiability of input theory, usually given in first-order logic. Since the negated conjecture is added to the input theory, the unsatisfiability result proves the conjecture. However, if the theory is satisfiable, many provers state only that

[1] Computing the stable model semantics, http://www.tcs.hut.fi/Software/smodels/, visited on June 2009
[2] The Dlv project, http://www.dbai.tuwien.ac.at/proj/dlv/, visited on June 2009
[3] Cmodels, http://www.cs.utexas.edu/users/tag/cmodels.html, visited on June 2009
[4] Assat, http://assat.cs.ust.hk/, visited on June 2009
[5] Clasp, http://www.cs.uni-potsdam.de/clasp/, visited on June 2009

it is satisfiable. Usually, there is no additional information which can be used as a clue to understand whether the conjecture is invalid or there is an error in the logical representation of the domain and the theorem. A model of the theory can provide valuable information in this sense. It may help to pinpoint a possible error in the representation or convince us that the conjecture is actually invalid. Unlike traditional theorem provers, model generation theorem provers can output a model when the input is satisfiable. We concentrated on Darwin[6], FM-Darwin[6], and Paradox[7] as model generation systems.

Current answer set solvers work on ground logic programs where there are no variables. It is highly cumbersome and error-prone process to represent problems with ground logic programs directly. Fortunately, users can conveniently write logic programs with variables which is, then, go through a pre-processing step using the systems called grounders. The grounding step converts a program with variables to a propositional one.

One rule of the program may have many variables. When the number of objects over which the variables of a rule range becomes relatively large, the grounder may generate a large number of propositional instances of the rule. This may increase the time spent by the grounder. Additionally, as the size of the propositional program becomes larger, the search process of the answer set solvers becomes more difficult which, in turn, may increase the search time. Therefore, the grounding step can be a bottleneck of ASP. Because of this, non-ground answer set computation has become one of the mostly stated issues for the future of ASP.[8]

The ability to complete a normal logic program with variables, and thus form a first-order theory, is the basis of this work. The first-order theory becomes the input to the model generation prover. We use model generation theorem provers as computational engines for ASP. When completion semantics [14] and answer set semantics [25] coincide [19, 4, 18] for a program, we can find its answer sets by searching the models of the corresponding first-order theory. In this way, we plan to eliminate the grounding step of ASP or perform it more intelligently using the model generation system. Additionally, regardless of grounding, we want to investigate the use of model generation systems as computational engines for ASP since they may display better performance than the current solvers. These constitute the main motivation of

[6] Darwin, http://combination.cs.uiowa.edu/Darwin/, visited on June 2009

[7] Paradox and Equinox, http://www.cs.chalmers.se/~koen/folkung/, visited on June 2009

[8] The community mentions this issue in the special session 'Celebrating 20 Years of Stable Model Semantics' at ICLP 2008. For example, V.Marek states in a more explicit way by 'old dream of non-ground solving still not realized'.

this work.

Answer set computation based on the relationship between completion and answer set semantics has been studied earlier and has led to successful answer set solvers such as Assat and Cmodels. These solvers complete grounded logic programs [37, 33]. Since the completed theory is propositional, they use SAT solvers to find the models. For this reason, they are called SAT-based answer set solvers. They have also managed to find answer sets in a sound way, even for non-tight programs on which completion semantics and answer set semantics differ. Our work does not include non-tight programs. The answer set computation proposed and experimented with here can be thought of as a lifting of the process of solvers, such as Assat and Cmodels, to the first-order level for tight programs.

A system called Completor[9] has been implemented for computing answer sets using model generation theorem provers. Completor handles all the necessary transformations that convert a normal logic program to a first-order theory suitable for the provers. Then, the model generation systems find the model of the resulting theory which corresponds to an answer set for a tight logic program.

We carried out several experiments in the following domains during this thesis in order to investigate the computation of answer sets using model generation theorem provers. All these problems are common benchmark problems for ASP.

- Graph theory. We used the Ramsey number problem as a benchmark problem.
- Combinatorics. In the community of automated reasoning the pigeon-hole problem is a well-known benchmark from the area of combinatorics.
- Algebra. We used the quasigroup existence problem from abstract algebra as a benchmark. In particular, QG2, QG5, and QG7 variants [40] are used.
- Planning. It is one of the most applied areas of ASP. We conducted experiments on the blocks world problem.

The empirical results demonstrate that the use of Completor and the theorem provers together, offers an effective way of computing answer sets. The runtime results of Paradox show that using Completor and Paradox is favorable compared to answer set solvers in Ramsey number,

[9] Completor, http://www.ceng.metu.edu.tr/~orkunt/completor/, visited on June 2009

pigeon-hole and quasigroup existence problems. Darwin and FM-Darwin has displayed weak performance on our benchmark problems, mainly due to axiomatic equality reasoning.

Our work does not claim that there is no need for grounding in ASP. Actually, it can be seen as a contribution to the dream of ASP without grounding. It may also increase the awareness of the ASP community regarding some recent theorem provers. The calculi and techniques used in model generation theorem provers can be beneficial to ASP without grounding.

The main contributions of this thesis can be listed as:

- A method for using model generation theorem provers as computational engines for answer set programming.
- Necessary transformations from normal logic programs to generate a first-order theory that is suitable for provers . Some of them are needed because of differences between first-order logic and answer set semantics. The theoretical research about correctness results of these transformations is also included (Theorems 4.3.7, 4.3.14 and 4.6.3).
- Empirical results showing that the use of model generation systems and Completor together, can be effective way of computing answer sets.
- Evaluating model generation theorem provers with benchmark problems. Some instances of these problems are difficult for the provers. Furthermore, Completor can be used as a tool to easily generate new benchmark tests by using logic programs belonging to various application domains as templates.

This thesis is organized as follows. Background information on answer set programming is provided in Chapter 2. Chapter 3 includes information on model generation theorem provers. In particular, it describes Darwin, Paradox and FM-Darwin. Transforming a normal logic program to a first-order theory and using model generation systems as computational engines for computing answer sets are described in Chapter 4. In Chapter 5, Completor is described and information on how to use it to form types of inputs for experiments is given. Chapter 6 includes the experimental results. In the final chapter, conclusions and future lines of research are given. In order to conduct experiments, a test environment has to be established by installing all the necessary solvers, provers, auxiliary programs and scripts. Appendix A describes how such an environment is established.

CHAPTER 2

ANSWER SET PROGRAMMING

Answer Set Programming is a declarative approach for solving search problems. Logic programming constitutes the foundation of ASP. In ASP, one represents a problem with a logic program whose models corresponds to solutions of the problem. Hence, unlike Prolog, it is not a proof-theoretical approach where you get solutions by answer substitutions.

Models of a logic program are specified by the answer set semantics (also known as stable model semantics) in ASP. Furthermore, these models are called *answer sets* of the program. Figure 2.1 depicts the flow of solving a problem in ASP.

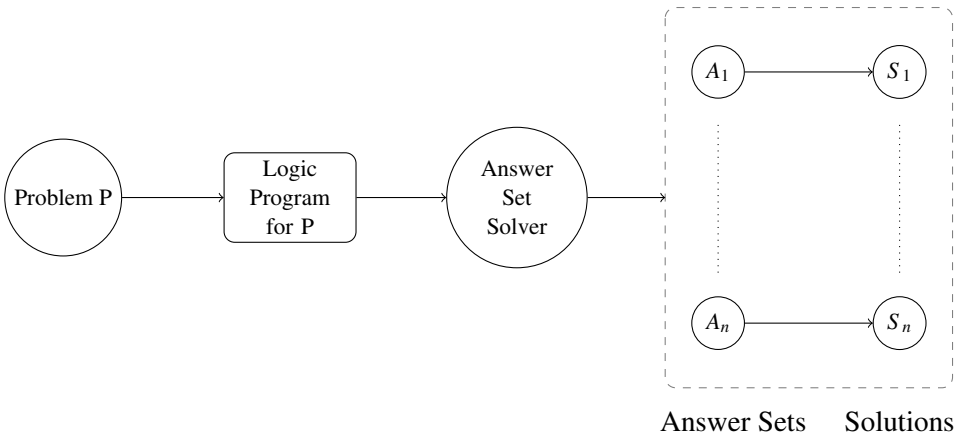

Figure 2.1: Solving a problem in Answer Set Programming

The origins of ASP is given in Section 2.1. In 1988, Gelfond and Lifschitz defined the answer set semantics in [24]. Section 2.2 defines the answer set semantics in detail.

The general structure of logic programs in ASP is usually composed of two main parts. In the generate part a candidate solution for the problem is populated. However, the candidate solution may not be a solution of the problem because of unsatisfied constraints. These constraints are represented in the test part of the logic program in order to test whether the candidate solution is a valid solution or not. This way of representing and solving a problem in ASP is called generate-and-test style. The example in Section 2.4 will make it more clear.

2.1 Basis of Answer Set Programming

The origins of ASP lies in logic programming. Logic programming [30] is one of the mostly used way of declarative programming. The reader is encouraged to consult [38] and [2] for detailed information on logic programming.

The procedural semantics of a logic program is defined by the SLDNF-resolution which is basically SLD-resolution [29, 30] augmented with negation as failure rule [14]. Prolog, a logic programming language, is based on SLDNF-resolution method. Using SLDNF-resolution, one can deduce negative facts. The ability to deduce negative information by a procedure is important, since it is a crucial property of nonmonotonic reasoning. Moreover, since many of the common-sense reasoning tasks involve nonmonotonic reasoning, the ability to reason with and deduce negative information directly effects the applicability of one's programming approach to wide range of domains.

Declarative semantics has to be defined in order to perform a theoretical analysis and prove soundness and completeness results of procedural semantics. Declarative semantics of a logic program is usually defined by a model which denotes all of the information that can be deduced by the program. In a way the declarative semantics defines the meaning of a program.

Although the procedural semantics of a logic program was defined clearly, the declarative semantics was not clear at first. In fact, the problem was to define the declarative semantics of negation as failure rule. For a positive program, where there is no negative literal in rules, the declarative semantics is defined by the least Herbrand model of the program. Fortunately, the least Herbrand model is unique for positive programs. However, for more general programs which have negative literals, it becomes difficult to designate a canonical model as the meaning. Several approaches had been proposed for defining the meaning of such programs.

In stratification [12, 3], a more constrainted subset of general programs has been considered. The concepts of perfect model [43] and well-founded model [23] have been introduced.

The stable model semantics [24] is also another approach to give meaning to logic programs with the negation as failure operator. This semantics is now known as answer set semantics and is the basis of ASP. Unlike the previous approaches, it does not designate a canonical model from all models of the logic program. In fact, there may be many intended models of the program. These models are called answer sets of the program. The fact that some programs may have more than one answer set was first criticized. However, the underlying idea behind answer set programming makes this property an advantage of answer set semantics. The computational problems may have more than one solution. The answer sets (possibly more than one) of the logic program representing the problem corresponds to the solutions of the problem.

2.2 Answer Set Semantics

Answer set semantics aims to give a declarative meaning to logic programs. First, we will define the answer set semantics for *propositional normal logic programs*. A propositional normal logic program is P a set of rules of the form:

$$h \leftarrow b_1, \cdots, b_m, \; not \; b_{m+1}, \cdots, not \; b_n \tag{2.1}$$

where h and all b's are propositional atoms. *not* is the negation as failure connective. A *literal* is either an atom or a negative atom which is preceded by the negation as failure connective *not*. Let r be the rule (2.1). The *head* of the rule $Head(r) = \{h\}$. The *body* of the rule r $Body(r) = \{b_1, \cdots, b_m, \; not \; b_{m+1}, \cdots, not \; b_n\}$, where the positive body $Body^+(r) = \{b_1, \cdots, b_m\}$ and the negative body $Body^-(r) = \{b_{m+1}, \cdots, b_n\}$.

Given a candidate set of atoms M which is a subset of the set of atoms occurring in the logic program P, the *reduct* of P relative to M, P^M is obtained by:

1. removing each rule whose body has a negative literal *not* b such that $b \in M$.

2. removing all negative atoms from the bodies of the remaining rules.

Formally, the reduct P^M is

$$\{Head(r) \leftarrow Body^+(r) \mid r \in P \text{ and } Body^-(r) \cap M = \varnothing\} \quad . \tag{2.2}$$

Note that for any propositional normal logic program, the reduct of it relative to some set of atoms is a program with no negation as failure connective. This kind of programs are called *definite programs* (also Horn programs) and have unique minimal Herbrand model. Let the unique minimal Hebrand model of P^M is M'. M' is defined as an answer set of P if $M' = M$.

Example 2.2.1 *Consider the program L consisting of the following rules.*

$$\begin{aligned} &a \leftarrow b,\ not\ d. \\ &d \leftarrow not\ a. \\ &c \leftarrow a. \\ &b. \end{aligned} \tag{2.3}$$

Let $A = \{a, b, c\}$. The reduct L^A consists of the following rules.

$$\begin{aligned} &a \leftarrow b. \\ &c \leftarrow a. \\ &b. \end{aligned} \tag{2.4}$$

Since the unique minimal Herbrand model of L^A is equal to A, A is an answer set of L. Let $B = \{d\}$. The reduct L^B is,

$$\begin{aligned} &d. \\ &c \leftarrow a. \\ &b. \end{aligned} \tag{2.5}$$

The unique minimal Herbrand model of L^B is $\{b, d\}$. Since it is not equal to the set B, B is not an answer set of L. A detailed definition and more examples can be found in [24].

The last rule in Example 2.2.1 has no body literals. These kind of rules are called *facts*. Additionally, there are rules where the head is empty. Those are called *constraints*. For instance, the rule

$$\leftarrow d, b. \tag{2.6}$$

is a constraint which states that no answer set of the program can have atoms d and b at the same time.

A normal logic program Π with variables is a set of rules of the form:

$$H \leftarrow B_1, \cdots, B_m, \ not\ B_{m+1}, \cdots, not\ B_n \tag{2.7}$$

where H and all B's are predicates without functions. For a program Π, $Obj(\Pi)$ denotes the set of object constants in Π. $Pred(\Pi)$ denotes the set of predicate constants in Π.

The definition of answer set semantics for a normal logic program with variables uses a pre-processing step called *grounding*. Grounding process transforms the program with variables to a propositional logic program by instantiating all rules of the original program in all possible ways using the object constants of the program. Thus, $Ground(\Pi)$ denotes the grounded program by instantiating every rule using the set $Obj(\Pi)$.

Example 2.2.2 *Consider the normal logic program N consisting of the following rules.*

$$\begin{aligned} &con(a, b). \\ &con(b, c). \\ &end(c). \\ &con(X, Y) \leftarrow con(Y, X),\ not\ end(X). \end{aligned} \tag{2.8}$$

The last rule is a non-ground rule since it includes the variables X and Y. The objects of N

$Obj(N) = \{a, b, c\}$. $Ground(N)$ consists of the following rules:

$$
\begin{aligned}
&con(a, b).\\
&con(b, c).\\
&end(c).\\
&con(a, a) \leftarrow con(a, a),\ not\ end(a).\\
&con(a, b) \leftarrow con(b, a),\ not\ end(a).\\
&con(a, c) \leftarrow con(c, a),\ not\ end(a).\\
&con(b, a) \leftarrow con(a, b),\ not\ end(b).\\
&con(b, b) \leftarrow con(b, b),\ not\ end(b).\\
&con(b, c) \leftarrow con(c, b),\ not\ end(b).\\
&con(c, a) \leftarrow con(a, c),\ not\ end(c).\\
&con(c, b) \leftarrow con(b, c),\ not\ end(c).\\
&con(c, c) \leftarrow con(c, c),\ not\ end(c).
\end{aligned}
\tag{2.9}
$$

Let M be a candidate set of atoms and a subset of the set of atoms occurring in $Ground(\Pi)$. M is an answer set of Π iff it is an answer set of the propositional program $Ground(\Pi)$. For the Example 2.2.2, the set $\{con(a, b),\ con(b, c),\ con(b, a),\ end(c)\}$ is an answer set of N, since it is an answer set of the grounded program (2.9).

2.3 Answer Set Solvers

Answer set solvers are systems that search for answer sets of input logic programs. The number of answer set solvers implemented has increased as ASP becomes widespread in use. The following is a list of well-known solvers.

- SMODELS. Smodels [49] can compute answer sets of normal logic programs with variables. The input program can use built-in functions like arithmetic and comparison functions for the convenience of the user. Grounding step is done by the system Lparse which is developed by the same research group. Lparse grounds the input program to form a propositional logic program. The search algorithm of Smodels work on the propositional program and is based on the Davis-Putnam procedure [16]. This proce-

dure is well-known in satisfiability checking (SAT) community. The selection heuristic used in the Davis-Putnam procedure is crucial for the search efficiency of Smodels [47].

There are some extended syntax which are supported by Smodels. Choice rules, cardinality constraints, weight rules and weight constraints help one to represent a problem domain in a concise and efficient way.

- DLV. Dlv [31] can compute answer sets of disjunctive logic programs. Disjunctive logic programs may have disjunction in the head of rules. Dlv has an internal grounder. The search procedure of Dlv is also based on the Davis-Putnam procedure. It has front-ends for planning, reasoning with description logics and external computation sources.

- CLASP. Clasp [22] is an answer set solver for normal logic programs. It uses Lparse as a grounder and supports extended features of Smodels like choice and weight rules. The search algorithm is based on conflict-driven clause learning, a technique used in satisfiability checking as an optimization. Clasp has also incorporated many successful optimization techniques used in satisfiability checkers.

- ASSAT. Assat [37] is a system that computes answer sets of a normal logic program with the help of SAT solvers. It is the first SAT-based ASP system. It uses Lparse as a grounder. The propositional program is first completed. Then, Assat gives the completed theory to the underlying SAT solver. Various SAT solvers can be used. Even when the two semantics (i.e., completion semantics and answer set semantics) do not coincide, Assat is capable of computing answer sets by adding additional constaint clauses called loop formulas.

- CMODELS. Cmodels [26] is a SAT-based answer set solver like Assat. It uses Lparse as a grounder. Just as Assat, Cmodels can also correctly compute answer sets for non-tight programs.

2.4 An Illustrative Example

We will use the Latin squares problem from algebra in order to show how a problem is represented and solved in ASP. A Latin square is an *nxn* square where every position of the square is filled by a number less than equal to *n*. Additionally, each row and column of the

square should not have the same number more than once. A popular logic puzzle, Sudoko, is a special case of the Latin square problem.

We can represent the square by its rows and columns. The rules (2.10) and (2.11) represent the rows and columns respectively. The rules (2.12) represent the numbers.

$$row(1). \quad \cdots \quad row(n). \tag{2.10}$$

$$col(1). \quad \cdots \quad col(n). \tag{2.11}$$

$$number(1). \quad \cdots \quad number(n). \tag{2.12}$$

The predicate $pos(X, Y, N)$ represents that the position (X, Y) in the square has the number N. Using the generate-and-test style of ASP, we can code the fact that we can assign any number to any position in the square to form the generate part. The rules (2.13) and (2.14) represents this fact. With the help of the *othernum* predicate and negation as failure connective, one is free to interpret a $pos(i, j, k)$ atom as true or false in an answer set. In this way, one can choose any number for a position in the square.

$$pos(X, Y, N) \leftarrow not\ othernum(X, Y, N),\ row(X),\ col(Y),\ number(N). \tag{2.13}$$

$$othernum(X, Y, N) \leftarrow not\ pos(X, Y, N),\ row(X),\ col(Y),\ number(N). \tag{2.14}$$

Next, we will represent the required constraints in order to have a valid Latin square. This will constitute the test part of the program. First, we have to eliminate the possibility that there are no unassigned positions. The rules (2.15) and (2.15) represent this constraint. The predicate $hasnum(X, Y)$ shows that the position (X, Y) has been assigned a number.

$$hasnum(X, Y) \leftarrow pos(X, Y, N),\ row(X),\ col(Y),\ number(N). \tag{2.15}$$

$$\leftarrow not\ hasnum(X, Y), row(X),\ col(Y). \tag{2.16}$$

The rule (2.17) represents that in a row a number can not be assigned more than once. Similarly, the rule (2.18) represents that in a column a number can not be assigned more than once. This completes the logic program for the problem of Latin squares.

$$\leftarrow pos(X, Y, N),\ pos(X, Y1, N),\ Y \neq Y1,\ row(X),\ col(Y),\ col(Y1),\ number(N). \tag{2.17}$$

$$\leftarrow pos(X, Y, N),\ pos(X1, Y, N),\ X \neq X1,\ row(X),\ row(X1),\ col(Y),\ number(N). \tag{2.18}$$

For a $4x4$ case, this logic program will have an answer set which includes the following set of *pos* atoms representing a valid Latin square.

$$\begin{aligned}\{\ & pos(1,1,3),\ pos(1,2,4),\ pos(1,3,2),\ pos(1,4,1),\\ & pos(2,1,4),\ pos(2,2,1),\ pos(2,3,3),\ pos(2,4,2),\\ & pos(3,1,1),\ pos(3,2,2),\ pos(3,3,4),\ pos(3,4,3),\\ & pos(4,1,2),\ pos(4,2,3),\ pos(4,3,1),\ pos(4,4,4)\ \}\end{aligned} \tag{2.19}$$

There are extended rules which can make the representation of the problem more concise. Smodels has extended the syntax of the normal logic programs with choice rules, weight rules and cardinality constraints. For instance, the choice rule,

$$\{\ pos(X,Y,N)\ \} \leftarrow row(X),\ col(Y),\ number(N). \tag{2.20}$$

can be used in place of rules (2.13) and (2.14). Rule (2.20) represents the fact that we can choose to assign any number to any position in the square. Moreover, cardinality constraints can be used in rule (2.20) to represent the fact that only one number can be assigned to a position. Thus, there will not be need to add the rules (2.15) and (2.16) if we use the following choice rule with cardinality constraints.

$$1\ \{\ pos(X,Y,N) : number(N)\ \}\ 1 \leftarrow row(X),\ col(Y). \tag{2.21}$$

The number 1 at the left of the head states that the number of *pos* atoms chosen to be true should be at least one. Similarly, the number 1 at the right of the head states that the number of *pos* atoms chosen to be true should be at most one.

2.5 SAT-Based Answer Set Solving

In 1978, Clark showed that negation as failure rule is sound according to the completion semantics of logic programs [14]. Basically, completing a program is interpreting the *if* statements in rules (i.e., $\leftarrow$ in a rule of a logic program) as *if and only if*. Thus, besides having sufficient conditions for an atom to be true, necessary conditions are also set. In this way, one can deduce negative information from the completion of a program. For programs with variables, the completion needs to be augmented with equational theory since it uses equality. The reader can find more detailed information on completion in Section 4.1.

After the answer set semantics had been defined, there were many researches studies on the relationship between the answer set semantics and completion semantics. In 1994, Fages defined a syntactic condition for logic programs when satisfied answer set semantics and completion semantics coincide. Hence, Herbrand models of the completion of a logic program are also its answer sets. Later, this condition is called *tightness* and more general tightness conditions are defined [18, 4]. Section 4.2 defines the tightness condition formally. The relationship between the two semantics has given rise to the idea of using SAT solvers to compute answer sets of logic programs. In this way, there will not be a need for dedicated answer set solvers.

Satisfiability checking is a well studied area in automated reasoning. With many optimizations applied in SAT solvers, it is possible to solve large problems. This forms another motivation for computing answer sets using SAT solvers. All the advances and know-how in satisfiability checking area will be automatically available for use when SAT solvers are used as black-box tools for computing answer sets.

Assat and Cmodels are SAT-Based answer set solvers. They both use Lparse as a grounder. After the grounding phase, they complete the propositional logic program to form a propositional theory. Then, the input for the SAT solvers is prepared from the resulting theory. They can use various SAT solvers, like zChaff, Sato and Relsat. The models of the completed propositional theory are answer sets of the program when it is tight.

The most significant feature of Assat and Cmodels is that they can also work for non-tight programs. When the input program is not tight, there may be models of its completion which are not answer sets. They both check whether the resulting model is an answer set or not. If it is not an answer set, a constraint is added to the completed theory in order to force that previous non-answer set model is no longer generated. This kind of formula added as a constraint is called a *loop formula*. More detailed information on forming loop formulas can be found in [37] and [26]. After adding a loop formula, both solver restart the search procedure by calling the SAT solver again to find a new model. This process continues until a model that is an answer set or the unsatisfiability result of the completed theory is found by the SAT solver. Figure 2.2 shows these steps for computing an answer set by Assat and Cmodels.

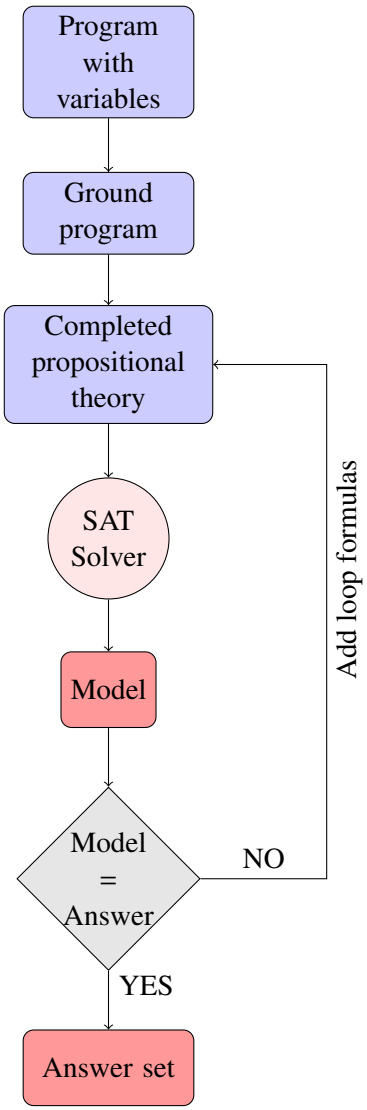

Figure 2.2: Computing an answer set by a SAT-based solver

CHAPTER 3

MODEL GENERATION THEOREM PROVERS

Theorem proving deals with finding the unsatisfiability result of a given theory in an automated way. This is directly related to the fact that first-order logic is semi-decidable but not decidable in general. A domain is represented with various axioms in the language of logic. The aim is to show that a conjecture about the domain is valid. Since many theorem provers are refutationally complete, we add the negated conjecture to the input theory. The expected result is the unsatisfiability of the input. This proves that the given conjecture is actually valid. However, if the prover outputs that the input is satisfiable,[1] it may be the case that the conjecture is invalid or there is an error in the logical representation of the domain. It is usually hard to pinpoint the reason of satisfiability result since traditional theorem provers just state that the input is satisfiable with no further information. A model that satisfies the given theory can be a valuable information in this sense. Besides, we can now add the conjecture not its negation to find a model which represents a valid case of the conjecture. With the capability of outputting a model when the input is satisfiable, one can use theorem provers in areas like diagnosis and planning. A solution of the problem can be captured from the model just like ASP.

Unlike traditional theorem provers, model generation theorem provers can output a model when the input is satisfiable. There are various calculi in which a model can be generated and presented. Generally, these calculi are in the class called instance based methods. The reader may refer to [28] for a review and comparison of instance based methods and [9] for a tutorial on the subject. Instance based methods search for a proof by maintaining a set of instances of input clauses and searching for satisfiability of these instances. In this thesis, we concentrated on model evolution calculus and finite model building.

[1] It may not be the case since the prover may not halt, since it is not complete in general

Model evolution calculus [10] is a lifting of propositional DPLL procedure [16, 15] to first-order level. It is refutationally complete and can also output a model when the input is satisfiable. Darwin is an implementation of model evolution calculus. In finite model building the prover tries to transform the input to propositional theory incrementally. At every step the prover calls a SAT solver to find a model whose domain size increase incrementally. Paradox is one of the finite model builders. FM-Darwin is also a finite model builder. It does not rely on SAT solvers, but uses Darwin as a first-order satisfiability checker instead. This thesis uses Darwin, Paradox and FM-Darwin as model generation theorem provers.

Sections 3.1, 3.2 and 3.3 describe the provers Darwin, Paradox and FM-Darwin respectively. The calculi which support these provers are also described in the respective sections.

3.1 Darwin

Almost all modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure. Successful techniques that are applicable in DPLL-based SAT, such as unit propagation, backjumping and learning, make modern SAT solvers very efficient. Darwin [8] is an implementation of the Model Evolution (ME) calculus [10], which lifts the DPLL procedure to the first-order level. The ME calculus also aims at keeping successful SAT techniques applicable in the lifted procedure.

DPLL procedure tries to find a model of the input set of propositional clauses. It updates a candidate model which is initially an empty set of literals. The derivation rules of the procedure try to add new literals to the candidate model in a deterministic way and delete the satisfied clauses from the input set. At some point of the derivation, the procedure cannot assert new literals anymore, but has to guess a truth value for an atom and add it to the candidate model. If a contradiction is found, the procedure backtracks to the last guess point and augments the candidate model with the complement of the literal which has been guessed at that guess point. When the input set of clauses becomes empty, DPLL procedure halts and the candidate model at hand satisfies all the input clauses.

Example 3.1.1 *Consider the set of clauses* $C = \{\neg p,\ p \vee \neg q \vee r,\ \neg q \vee \neg r,\ q \vee r\}$. *The DPLL procedure can find the model* $\{\neg p, \neg q, r\}$, *which satisfies the clauses C, with the following derivation.*

$\{\cancel{\neg p},\ \cancel{p} \vee \neg q \vee r,\ \neg q \vee \neg r,\ q \vee r\}$

$\Downarrow$ Assert $\neg p$

$\{\cancel{\neg q} \vee r,\ \cancel{\neg q} \vee \neg r,\ \cancel{q \vee r}\}$

$\Downarrow$ Guess q

$\{r,\ \neg r\}$

$\Downarrow$ Contradiction, backtrack

$\{\cancel{\neg q \vee r},\ \cancel{\neg q \vee \neg r},\ \cancel{q} \vee r\}$

$\Downarrow$ Guess $\neg q$

$\{r\}$

$\Downarrow$ Assert r

$\{\ \}$

Note that after the application of Assert and Guess derivation rules, some of the input clauses are simplified or deleted by the Subsume and Resolve derivation rules. For example, after asserting $\neg p$, the clause $\neg p$ is subsumed and the clause $p \vee \neg q \vee r$ is resolved to $\neg q \vee r$. The first application of the Guess rule has added q to the candidate model, which leads to a contradiction. The procedure has to backtrack to the last guess point and add the literal $\neg q$ to the candidate model this time.

The search procedure of Darwin lifts the model generation process of DPLL procedure to the first-order level [8]. It tries to find a Herbrand model of the input set of clauses, if one exists. However, this time the input may be first-order clauses. The ME calculus manages a structure called *context* as a candidate model instead of a simple set of propositional literals.

A *context* Λ is a finite representation of a Herbrand interpretation I_Λ. It is a set composed of ground and non-ground literals. An initial context represents an interpretation which falsifies all the atoms in the input. When there is a clause which is not satisfied by the Herbrand interpretation represented by the current context, the derivation rules repair the context by adding literals. If all the clauses are satisfied, then the context actually represents a Herbrand model of the input theory. The ME calculus can also detect a situation where there are no possible repairs of the context. This implies the unsatisfiability of the input theory.

Example 3.1.2 *Consider the context* $\Lambda = \{p(x, y),\ \neg q(x, x)\}$ *where x and y are variables. Let*

the constants a and b be part of the input signature. The Herbrand interpretation I_Λ satisfies the atoms $p(a,a)$, $p(a,b)$, $p(b,a)$. $p(b,b)$, $\neg q(a,a)$ and $\neg q(b,b)$.

There are two kinds of variables defined in the ME calculus; *universal variables* (variables) and *parametric variables* (parameters). Context literals are either *universal* (i.e., they contain only variables) or *parametric* (i.e., they contain only parameters), but not mixed. Each literal of a context, either universal or parametric, stands for all its ground instances, which, as a whole, compose a Herbrand interpretation. Roughly, a parametric literal L represents all its ground instances except those that are instances of another context literal $\neg L'$, where L' is an instance of L.[2] However, a universal literal stands for all its ground instances with no exceptions.

Example 3.1.3 *The following is an example from a presentation of the ME calculus.*[3] *Let $\Lambda_1 = \{p(u,v), \neg p(u,u), p(f(u), f(u))\}$ where u and v are parameters. The Herbrand interpretation I_{Λ_1} satisfies every instance of $p(u,v)$ except that u and v are the same. It satisfies every instance of $\neg p(u,u)$ except those of the positive literal $p(f(u), f(u))$. Figure 3.1 depicts the instances of p that are produced by Λ_1 graphically. If we change the negative literal of Λ_1 to a universal one, we get $\Lambda_2 = \{p(u,v), \neg p(x,x), p(f(u), f(u))\}$ where x is a variable. The universal literal $\neg p(x,x)$ imposes a strict restriction such that when the arguments of p are the same, the context produces it negatively. Since the literal $p(f(u), f(u))$ is in contradiction with this restriction, Λ_2 is contradictory. The ME calculus performs contradiction checks during the derivation in order to work with a non-contradictory context.*

There are two important situations which the ME calculus should detect during the derivation. The first is the case in which the current context falsifies an input clause. The Split rule applies when this situation holds. This rule repairs the context by adding a literal from the falsified instance of the clause in order to satisfy it. The Split rule is the corresponding rule to DPLL's Guess rule. The other important situation is the case in which the current context permanently falsifies a clause, no matter how the context is repaired [10]. This implies that there is no possible repair of the context. The Close rule detects this situation. After the application of the Close rule, the calculus tries to backtrack to the most recent application of the Split rule.

[2] The precise definition of how a Herbrand interpretation is induced from a context is given in [10].

[3] The Model Evolution Calculus, http://users.rsise.anu.edu.au/~baumgart/slides/MEGoeteborg.pdf, visited on June 2009

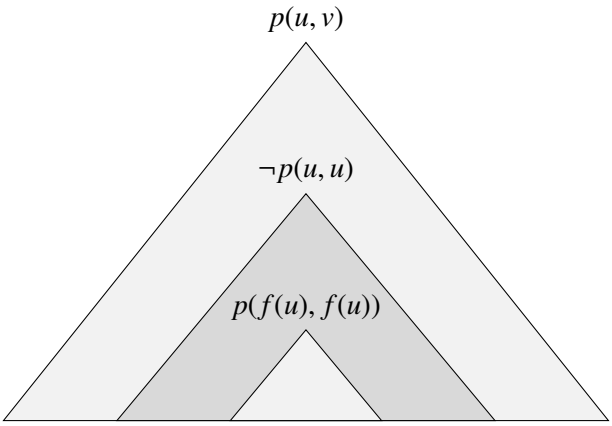

Figure 3.1: Instances of the predicate p produced by the context Λ_1

The ME calculus has other rules for a fair lifting of DPLL procedure to the first-order level. There are first-order counterparts of the Assert, Subsume and Resolve rules. The reader may refer to [10] for detailed definitions of the derivation rules.

Example 3.1.4 *The following example is a simple set of clauses which is satisfiable.*

$$p(x) \vee t(x) \tag{3.1}$$

$$\neg t(a) \tag{3.2}$$

$$q(x) \vee r(x) \vee \neg p(x) \tag{3.3}$$

$$\neg r(x) \vee \neg q(x) \tag{3.4}$$

Figure 3.2 shows Darwin's derivation of a model of this input. The derivation start with an empty context. Assert derivation rule uses the clause (3.2) to add the $\neg t(a)$ to the context. The new context and $p(a) \vee t(a)$, an instance of the clause (3.1) cause Darwin to fire the Assert rule again and add $p(a)$ to the context. Note that Assert rule is the first-order counterpart of the Assert rule in propositional DPLL procedure as shown in Example 3.1.1. Then, Darwin detects that an instance of the clause (3.3) is not satisfied by the current context. The Split rule choose the q predicate from the unsatisfied instance $q(a) \vee r(a) \vee \neg p(a)$ and branches the derivation by $q(a)$ in left and $\neg q(a)$ in right.[4] The derivation continues along the left branch

[4] The Split rule of ME calculus tries to choose a literal from the remainder of the unsatisfied clause instance. $q(a) \vee r(a)$ is the remainder part since the disjunct $\neg p(a)$ is unsatisfied readily by the $p(a)$ literal in the context.

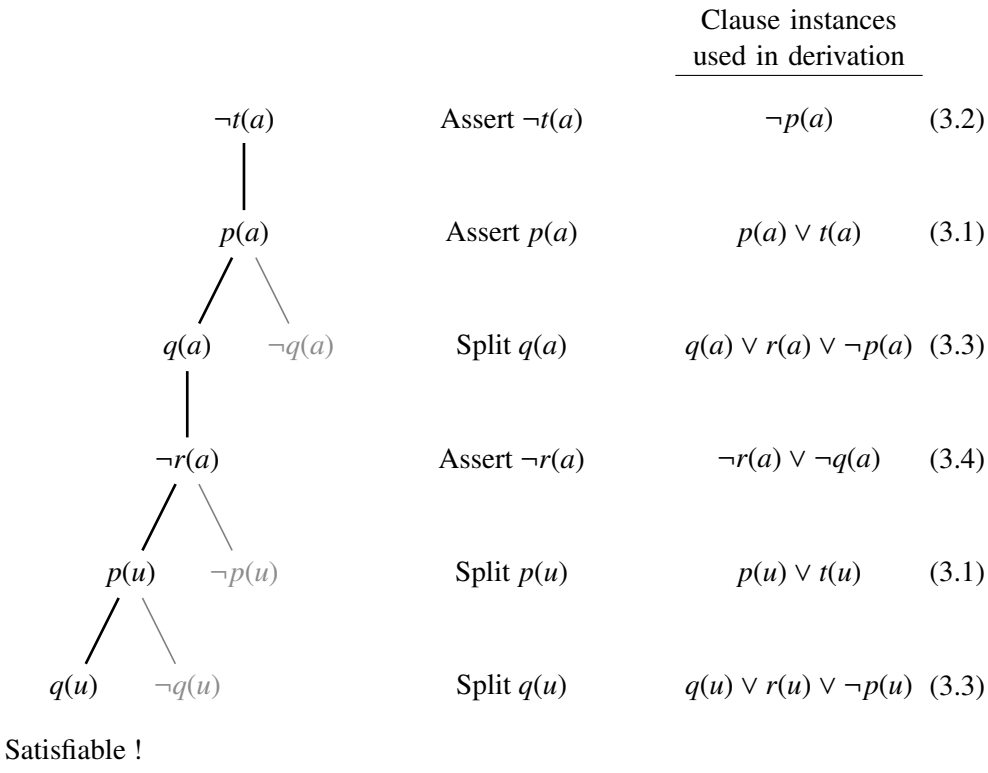

Figure 3.2: A derivation of Darwin / ME calculus

which is shown by bold lines in Figure 3.2. Last two applications of Split rule add parametric literals $p(u)$ and $q(u)$ to the context in order to satisfy all instances of the clauses (3.1) and (3.3). Then, the context at hand satisfies all clauses in the input. Darwin states this situation and outputs that the input is satisfiable. The Hebrand interpretation represented by the context at the end of the derivation,

$$\{\neg t(a),\ p(a),\ q(a),\ \neg r(a),\ p(u),\ q(u)\} \tag{3.5}$$

is a model for the input.

Although Darwin's proof procedure is refutationally complete, it is not complete for finite satisfiability (i.e., it is not guaranteed to find a finite model, if one exists) [8]. When the input has an infinite Herbrand universe, the proof procedure may become stuck in a endless derivation and not halt, even if the input has a finite model. However, Darwin is a decider for Bernays-Schönfinkel formulas [10]. The clausal form of these formulas has no functions, and thus has a finite Herbrand universe. In the scope of our work, a state of not being complete for finite satisfiability has an implication such that when the clausal form of the completed logic

program has Skolem functions, Darwin may not halt. Hence, we have transformations which eliminate the Skolem functions and create formulas in Bernays-Schönfinkel form in order to guarantee that Darwin halts with an answer. See Section 4.3 for a detailed discussion of this implication.

3.2 Paradox

If there is an interpretation which satisfies a first-order theory, one can easily construct another interpretation from a new domain of objects and a bijection from the domain of the first interpretation to the new one. The constructed interpretation is also a model of the theory [13]. This fact implies that the actual members of the domain are not important in forming a model. An arbitrary set with a sufficient number of objects can be used as a domain since only the size of the domain matters.

Example 3.2.1 *Let the set of formulas L be* $\{r(1).,\ \neg p(1).,\ p(f(x)) \leftarrow r(x).\}$. *The model* $\{r(1), \neg p(1), p(f(1))\}$ *satisfies L. The interpretation* I_1 *shown below generates this model with the domain* $\{a, b\}$. I_1 *maps the terms* 1 *and* $f(1)$ *to the domain objects a and b, respectively. One can construct another interpretation* I_2 *with a new domain from the interpretation* I_1 *and the bijection* π*, which maps the domain elements a and b to* ♦ *and* ★*, respectively. The interpretation* I_2 *also satisfies L.*

$I_1(r)(a) = \mathrm{T}$	$I_2(r)(\blacklozenge) = \mathrm{T}$
$I_1(r)(b) = \mathrm{F}$	$I_2(r)(\bigstar) = \mathrm{F}$
$I_1(p)(a) = \mathrm{F}$	$I_2(p)(\blacklozenge) = \mathrm{F}$
$I_1(p)(b) = \mathrm{T}$	$I_2(p)(\bigstar) = \mathrm{T}$
$I_1(f)(a) = b$	$I_2(f)(\blacklozenge) = \bigstar$
$I_1(f)(b) = b$	$I_2(f)(\bigstar) = \bigstar$
$I_1(1) = a$	$I_2(1) = \blacklozenge$

Paradox [13], a Mace-style finite model builder [40], utilizes the above fact about the unimportance of the actual members of a domain to find a model of the input set of first-order clauses. It searches for a model with a finite domain by enumerating the finite domains with

sizes in increasing order such as $\{1'\}, \{1', 2'\}, \{1', 2', 3'\}, \cdots$ and so on. The builder incrementally checks for each finite domain starting with $\{1'\}$ if there exists an interpretation that satisfies the input clause set. The domain size is increased until a model is found. The actual search for a model is achieved by instantiating the input clauses for each domain and running a SAT solver on the propositional theory.

The reader may refer to [13] for a detailed description of how input clauses are instantiated for each domain during the search process.

The instantiation of input clauses starts with a process called *flattening* [13, 40, 53]. Flattening a clause makes it flat. Flat clauses have only flat literal (also called shallow literals). The predicate of a flat literal has only variables as arguments. When an argument of a literal has a term in the form of a constant or a function, the clause having the literal is rewritten by the following rewrite rule:

$$C[t] \rightsquigarrow t \neq x \vee C[x]$$

where t is the term and x is a fresh variable. Equality is handled as a binary predicate. Thus, equality literals should also be flat. A flat equality literal has one of the following forms:

$$f(x_1, \cdots, x_n) = y \text{ or } f(x_1, \cdots, x_n) \neq y$$

$$x = y \text{ or } x \neq y$$

where $x, x_1, \cdots, x_n, y$ are variables.[5] Since an object constant is a nullary function, one argument of the equality predicate can be an object constant. If a function is not flat, it is also rewritten by the above rewrite rule.

After flattening the clauses in Example 3.2.1, we get the following flat clauses.

$$\begin{aligned} &r(x) \vee x \neq 1 \\ &\neg p(x) \vee x \neq 1 \\ &p(y) \vee \neg r(x) \vee f(x) \neq y \end{aligned} \tag{3.6}$$

The next step is to form a propositional theory from flat clauses. This is done incrementally until a model is found as described before. Thus, Paradox first grounds flattened clauses for domain size 1. If a model can not be found by the SAT solver, it grounds flattened clauses for domain size 2. In this instantiation process, variables of the flat clause are substituted

[5] This condition is slightly different from the one in [13] where literals of the form $x \neq y$ are eliminated by another rewrite rule. Clauses having these literals can be simplified during instantiation process.

by domain elements in every possible way. For a domain size n, a flat clause will have n^k number of propositional clauses where k is the number of distinct variables in the clause. Additionally, totality and uniqueness axioms are added to the propositional theory for defining the functions used in the input. The totality axiom states that a function's value is one of the domain elements, but nothing else. The uniqueness axiom states that a function's value is unique for the same arguments.

The equality predicate has a fixed interpretation in MACE-style finite model building. The model should assign the literal $d_1 = d_2$ as true only when it assigns terms d_1 and d_2 the same domain element. For the other cases, it should be false. The builder can eliminate or simplify a clause instance containing equality literals during the instantiation phase. The capability of reasoning with equality is necessary for our work, since the completion of a logic program with variables introduces equality predicates.

After instantiating the set of flat clauses (3.6) for domain $\{1'\}$, we get the following set of propositional clauses.

$$\begin{aligned} &r(1') \vee 1' \neq 1 \\ &\neg p(1') \vee 1' \neq 1 \\ &p(1') \vee \neg r(1') \vee f(1') \neq 1' \\ &1 = 1' \\ &f(1') = 1' \end{aligned} \tag{3.7}$$

The last two clauses in (3.7) are the totality axioms for the object 1 and the function f respectively. The uniqueness axiom for f is eliminated since there is only one element in the domain. Paradox gives (3.7) to SAT solver as input. Clause set (3.7) is unsatisfiable. After the SAT solver finds this result, Paradox instantiates (3.6) again, but this time for domain $\{1', 2'\}$. The following is the resulting set of propositional clauses after this instantiation.

$$\begin{aligned} &r(1') \vee 1' \neq 1 \\ &r(2') \vee 2' \neq 1 \end{aligned} \tag{3.8}$$

$$\begin{aligned} &\neg p(1') \vee 1' \neq 1 \\ &\neg p(2') \vee 2' \neq 1 \end{aligned} \tag{3.9}$$

$$
\begin{aligned}
&p(1') \vee \neg r(1') \vee f(1') \neq 1' \\
&p(1') \vee \neg r(2') \vee f(2') \neq 1' \\
&p(2') \vee \neg r(1') \vee f(1') \neq 2' \\
&p(2') \vee \neg r(2') \vee f(2') \neq 2'
\end{aligned} \tag{3.10}
$$

$$
\begin{aligned}
&f(1') \neq 1' \vee f(1') \neq 2' \\
&f(2') \neq 1' \vee f(2') \neq 2'
\end{aligned} \tag{3.11}
$$

$$
1 = 1' \vee 1 = 2' \tag{3.12}
$$

$$
\begin{aligned}
&f(1') = 1' \vee f(1') = 2' \\
&f(2') = 1' \vee f(2') = 2'
\end{aligned} \tag{3.13}
$$

The clauses (3.11) are uniqueness axioms for f. The clauses (3.12) and (3.13) are totality axioms for 1 and f respectively. The SAT solver can find a model for the input (3.8) - (3.13). From the resulting propositional model, Paradox builds the same finite model which is shown in Example 3.2.1. In this building phase, Paradox interprets atoms in the propositional model found by the SAT solver. For instance, if the atom $p(2')$ is true in the propositional model, $I(p)(2') = \mathrm{T}$ holds in the finite model. Similarly, if the atom $f(1') = 2'$ is true, $I(f)(1') = 2'$ holds in the finite model.

Paradox has many optimizations that enhance the original Mace-style finite model building method [13]. In order to reduce the number of instances of a clause, a clause splitting technique is used, which reduces the number of variables in a clause. Another optimization involves breaking symmetric models. Many symmetric models in finite model building make the job of the SAT solver harder, since it considers them as different models. A model assigns each constant a domain element. Any permutation of this assignment will generate symmetric models. For instance, in Example 3.2.1, the interpretation I_1 maps the constant 1 to the domain element a and the term $f(1)$ to b. However, a symmetric model can be formed by mapping 1 and $f(1)$ to b and a respectively. Many symmetric models cause an increase in the search space of the SAT solver. Paradox adds some extra clauses to the propositional theory in order to reduce symmetries [13].

The SAT solver which is embedded in Paradox has incremental solving capability. It does not start from scratch for every domain size [13]. For example, after the solver has searched for a model with a finite domain of size 2, the propositional instance of the problem for a

finite model of size 3 will not be prepared from the very beginning. Instead, Paradox keeps the instances of the clauses for size 2 and adds some other propositional clauses which are needed for size 3. Also, the SAT solver has the capability in which the conflict clauses that have been learned during an unsuccessful search for a model in previous iterations can be reused for the next domain sizes.

Paradox also uses sort inference as another optimization. A sort can be viewed as a type in typed first-order logic. The quantification of the variables in a clause should be over its assigned sort. The same domain elements can be used for interpreting functions and predicates that are of different sorts. This will decrease the size of the model. Additionally, there may be fewer instances of the clauses respecting sorts compared to the unsorted case. This will reduce instantiation time and ease model finding. Paradox infers the sorts of the problem by analyzing its first-order logic encoding [13]. It uses the fact that same variables in a clause should be of the same sort. Also, the arguments of the equality symbol should be of the same sort. In the scope of our work, Section 4.3 describes how to take advantage of sort inference optimization.

Unlike Darwin, Paradox is complete for finding models with finite domains, but not refutationally complete in general. However, it is complete for first-order input in the Effectively Propositional (EPR) class [13]. EPR class problems have no functions of arity greater than 0.[6] The number of the elements in the largest sort inferred for an EPR problem is the limit for its satisfiability. There can be no models with a domain size greater than this limit [13]. If no model can be found of size up to that number, Paradox decides that the input is unsatisfiable. In our work, while working with the completion of a logic program or the clausal form of its completion, the input of Paradox may not be in EPR class. Thus, if the logic program has no answer sets, Paradox may not halt since it is not refutationally complete in general. However, the transformation for elimination of Skolem functions in the clausal form of the completion of a logic program will generate formulas in EPR class (see Section 4.3). Thus, Paradox will be guaranteed to halt for that type of input, even if the corresponding logic program has no answer sets.

[6] The Bernays-Schönfinkel class of formulas is also referred to as EPR class.

3.3 FM-Darwin

FM-Darwin[7] is another Mace-style finite model builder [7]. However, while Paradox transforms the problem to a propositional satisfiability problem, FM-Darwin transforms it to a satisfiability problem of function-free clause logic without equality. The ME calculus and Darwin are deciders for that class of first-order logic [10]. FM-Darwin uses Darwin as an engine to solve the transformed satisfiability problem. Note that FM-Darwin implements the Mace-style finite model building without grounding, since Darwin can work on the first-order level. FM-Darwin performs optimizations similar to those in Paradox for reducing symmetries. It also has the sort inference optimization.

Just like Paradox, FM-Darwin is complete for finding models with finite domains, but not refutationally complete in general. It is a decider for EPR class problems. Hence, the implications related with these properties that are mentioned for Paradox also apply to FM-Darwin.

[7] FM-Darwin comes embedded in Darwin. It can be used by running Darwin with some command-line arguments.

CHAPTER 4

LIFTING SAT-BASED ANSWER SET COMPUTATION TO FIRST-ORDER LEVEL

SAT-based answer set computation is described in Section 2.5. The basic idea of SAT-based ASP is to use SAT solvers instead of dedicated answer set solvers for computing answer sets. With the advances and optimizations in satisfiability checking, SAT solvers can be used for solving large real world problems. This is one of the practical motivations of SAT-based answer set computation. Assat and Cmodels are successful SAT-based answer set solvers.

In order to use SAT solvers, a logic program has to be converted to set of clauses in propositional logic. Completion [14] is the process of transforming a logic program to a propositional theory. Section 4.1 defines the completion process formally. Although all answer sets of a logic program are also models of the program's completion, the converse does not hold in general. It may be the case that for a program, some of the models of its completion are not its answer sets. The theory behind SAT-based answer set computation reflects the relationship between completion and answer set semantics. When the two semantics coincide, it is safe to use a SAT solver on the completed program. Fortunately, Assat and Cmodels can also work when the two semantics do not coincide.[1] The programs for which the two semantics are equal are called *tight programs* [19, 18]. The sufficient conditions for a program to be tight (i.e., tightness condition) are defined in Section 4.2.

SAT-based solvers use a grounder to form a propositional logic program before doing completion. As the size of the propositional program becomes larger, it may become difficult for the SAT solver to find models of the program's completion. Thus, like in the case of dedicated answer set solvers, grounding can become a bottleneck of SAT-based answer set solvers.

[1] They add loop formulas to the propositional theory in order to eliminate models which are not answer sets.

Model generation theorem provers can find a model of given first-order theory when it is satisfiable. They can be used as computational engines for computing answer sets just like SAT solvers. A given normal logic program with variables can be transformed to a first-order theory. This theory can be used as an input of model generation theorem provers in order to compute a satisfying model. For tight programs, the resulting model is an answer set. Naturally, if the prover finds the unsatisfiability of the input, we can conclude that the initial logic program has no answer sets. Note that in this process we have not applied grounding. However, a model generation theorem prover may ground the input internally. For instance, Paradox applies incremental grounding for building a finite model (see Section 3.2). The main motivation of this work is to eliminate the grounding step of ASP or perform it more intelligently using model generation systems. Some of the experiments in Chapter 6 has positive results showing that this way of computing answer sets can indeed be advantageous compared to using other answer set solvers, both dedicated and SAT-based ones.

When compared to SAT-based answer set computation, the proposed process can be seen as a lifting of the flow of SAT-based answer set solving to the first-order level for tight programs. Figure 4.1 depicts this lifted process compared to the SAT-based answer set solving shown by Figure 2.2.

In addition to the completion of the program, some extra axioms and transformations are needed in order to compute the answer sets correctly. This is due to some restrictions and completeness results of the model generation theorem provers that are used. All transformations which are necessary for generating a first-order theory from a normal logic program with variables in order to compute an answer set are defined and discussed in Section 4.3. Types of input theories, which are formed by these transformations, and completeness properties of model generation theorem provers for these input types are discussed in Section 4.4. Section 4.5 describes the way of generating a first-order input theory which is suitable for sort optimization used by finite model builders. Section 4.6 states an interesting theorem which says that there is no need for adding a domain closure axiom when constructing input theories for domain-restricted logic programs.

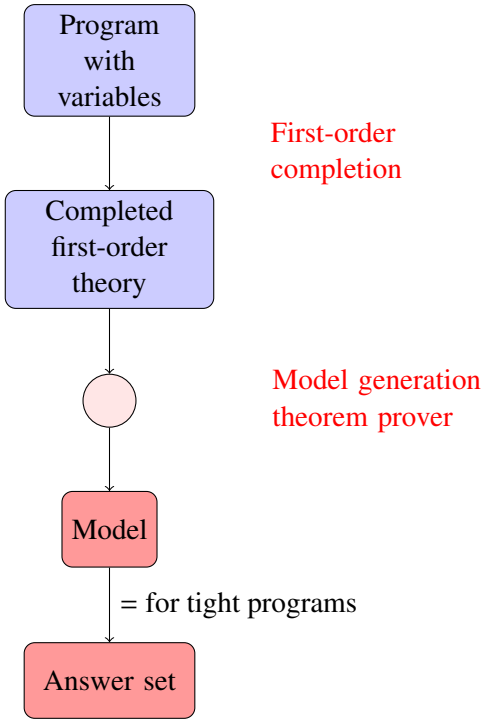

Figure 4.1: Computing an answer set by model generation theorem provers

4.1 Completion

A rule of a normal logic program Π with variables is of the form:

$$H \leftarrow B_1, \cdots, B_m, \; not \; B_{m+1}, \cdots, not \; B_n \tag{4.1}$$

where H and all B's are function-free atoms. Let the head H be of the form $h(\overrightarrow{t})$ (i.e., an instance of the predicate h with arity s) and $\overrightarrow{t}$ be a tuple of s terms. The *general form of a rule* of the form (4.1) is defined as

$$h(\overrightarrow{u}) \leftarrow \exists \overrightarrow{y} \; (B_1 \wedge \cdots \wedge B_m \wedge \neg B_{m+1} \wedge \cdots \wedge \neg B_n \wedge u_1 = t_1 \wedge \cdots \wedge u_s = t_s) \tag{4.2}$$

where $\overrightarrow{u} = \langle u_1, \cdots, u_s \rangle$ is a tuple of distinct and fresh variables and $\overrightarrow{y}$ is a tuple of variables in the rule. If there are k rules $\{r_1, \cdots, r_k\}$ whose head is an instance of the predicate h, $Comp(h)$, the completion of the predicate h, denotes the first-order formula

$$\forall \overrightarrow{u} \left(h(\overrightarrow{u}) \; \Leftrightarrow \; \bigvee_{1 \leq i \leq k} ConjPart(r_i) \right) \tag{4.3}$$

where $ConjPart(r_i)$ is the antecedent of the general form of the rule r_i. The $ConjPart$ of a rule whose general form is of the form (4.2) is

$$\exists \overrightarrow{y} \ (B_1 \wedge \cdots \wedge B_m \wedge \neg B_{m+1} \wedge \cdots \wedge \neg B_n \wedge u_1 = t_1 \wedge \cdots \wedge u_s = t_s) \ . \tag{4.4}$$

If there are no rules with h in the head, then the completion of the predicate h is

$$\forall \overrightarrow{u} \ (h(\overrightarrow{u}) \Leftrightarrow \bot) \ . \tag{4.5}$$

The *completion of a program* Π is the set of the completions of all the predicates in Π [14].

$$Comp(\Pi) = \{ \ Comp(h) \mid h \in Pred(\Pi) \ \} \tag{4.6}$$

Example 4.1.1 *Let the logic program* Π_1 *be the set of following rules:*

$$\begin{array}{l} con(a, b) \\ con(b, c) \\ con(c, c) \\ end(c) \\ sep(X) \leftarrow con(X, Y), \ not \ end(Y) \end{array} \tag{4.7}$$

$Comp(\Pi_1)$ is the following formulas of first-order logic.

$$\begin{array}{ll} \forall x & (end(x) \Leftrightarrow (x = c)) \\ \forall xy & (con(x, y) \Leftrightarrow ((x = a \wedge y = b) \vee (x = b \wedge y = c) \vee (x = c \wedge y = c))) \\ \forall x & (sep(x) \Leftrightarrow \exists y \ (con(x, y) \wedge \neg end(y))) \end{array} \tag{4.8}$$

The general form of a rule with variables in the head may include fresh variables and equalities between these new variables and ones in the original rule. These equalities can be eliminated and an equivalent and simplified formula can be formed as the completion of the rule. For instance, the last formula of (4.8) is a simplified formula which is equivalent to

$$\forall u \quad (sep(u) \Leftrightarrow \exists xy \ (con(x, y) \wedge \neg end(y) \wedge u = x)) \ . \tag{4.9}$$

4.2 Tight Logic Programs

A condition for the equivalence of completion and answer set semantics has been set first by Fages' theorem [19]. This is a syntactic property and called tightness in [34]. [2] For a *tight*

[2] In [19], programs that have this property are called positive-order-consistent programs.

program, there exists a function λ from ground atoms to non-negative integers such that for every rule

$$h \leftarrow b_1, \cdots, b_m, \; not \; b_{m+1}, \cdots, not \; b_n \tag{4.10}$$

in the grounded program, $\lambda(h) > \lambda(b_1), \cdots, \lambda(b_m)$ (i.e., the value assigned to h should be greater than all the values assigned to positive body literals) [4]. Fages' theorem states that, for a tight program, the Herbrand models of the completion of the program coincide with its answer sets.

Let M be a Herbrand model and A be an answer set of a logic program Π. M coincides with A, if it interprets the predicates $Pred(\Pi)$ same as A does. There may be auxiliary atoms in the Herbrand model. Formally, *M coincides with A* if $M \supseteq A$.

Consider the program Π_1 in Example 4.1.1. Π_1 is tight since we can find a function λ satisfying the tightness condition with the following definitions:

$$\lambda(con(\_, \_)) = 0$$

$$\lambda(end(\_)) = 0$$

$$\lambda(sep(\_)) = 1$$

λ assigns 0 to every ground instance of the predicates *con* and *end* and 1 to every ground instance of the predicate *sep*. The value assigned to a ground *sep* atom should be greater than the value assigned to a ground *con* atom, since *sep* instances are head atoms and *con* instances are positive body atoms of ground instances of the last rule in (4.7).

Since Π_1 is tight, Herbrand models of its completion coincide with its answer sets. Indeed, the model $\{con(a, b), \; con(b, c), \; con(c, c), \; end(c), \; sep(a)\}$ satisfies the completion (4.8) and is an answer set of Π_1.

Example 4.2.1 *Let the logic program Π_2 be composed of the rules of Π_1 in Example 4.1.1 and the following two additional rules.*

$$\begin{aligned} tc(X, Y) &\leftarrow con(X, Y) \\ tc(X, Y) &\leftarrow con(X, Z), \; tc(Z, Y) \end{aligned} \tag{4.11}$$

For Π_2, the completion $Comp(\Pi_2)$ is the set of formulas in (4.8) augmented with the following formula.

$$\forall xy \quad (tc(x, y) \Leftrightarrow con(x, y) \vee \exists z \; (con(x, z) \wedge tc(z, y))) \tag{4.12}$$

The ground instances of the last rule in (4.11) includes the rules,

$$\begin{aligned} tc(c,a) &\leftarrow con(c,c),\ tc(c,a) \\ tc(c,b) &\leftarrow con(c,c),\ tc(c,b) \\ tc(c,c) &\leftarrow con(c,c),\ tc(c,c) \end{aligned} \tag{4.13}$$

For each rule in (4.13), the head atom is also in the positive part of the body. This fact makes finding a function that satisfies the tightness condition impossible. For instance, the tightness condition necessitates that $\lambda(tc(c,a)) > \lambda(tc(c,a))$. However, there is no such λ. Thus, Π_2 is not tight. Actually, the model (4.14) is a Herbrand model of $Comp(\Pi_2)$, but not an answer set of Π_2 because of the atom $tc(c,b)$.

$$\begin{aligned} \{\ & con(a,b),\ con(b,c),\ con(c,c),\ end(c),\ sep(a), \\ & tc(a,b),\ tc(b,c),\ tc(a,c),\ tc(c,c),\ tc(c,b)\ \} \end{aligned} \tag{4.14}$$

There is a generalization of the tightness condition in [18, 4], which is called *tightness on a set of literals*. Some logic programs are tight on each of the model of their completion. For such a program, Hebrand models of its completion coincide with its answer sets, even if it is not tight according to the original definition. The program in Section 6.4 is tight on models of its completion [4].

4.3 Transforming Logic Programs to First-order Theories for Model Generation Theorem Provers

This section defines the necessary transformations used in the lifted flow shown in Figure 4.1 thoroughly. The following logic program will be used as an illustrative example throughout the section.

Example 4.3.1 *Let the logic program $\mathcal{P}$ be*

$$\begin{aligned} & q(2). \quad r(a,1). \quad r(b,2). \\ & p(X) \leftarrow r(X,Y),\ not\ q(Y). \end{aligned} \tag{4.15}$$

The program $\mathcal{P}$ is tight, since we can find a function λ such that $\lambda(q) = 0$, $\lambda(r) = 0$ and $\lambda(p) = 1$.[3] *$\mathcal{P}$ has only one answer set, which is $\{q(2),\ r(a,1),\ r(b,2),\ p(a)\}$.*

[3] The predicate names in the equations represent all of their ground instances.

After completing $\mathcal{P}$, the first-order theory $Comp(\mathcal{P})$ is found as:

$$\begin{aligned} &\forall x \quad (q(x) \Leftrightarrow (x = 2)), \\ &\forall xy \quad (r(x,y) \Leftrightarrow ((x = a \wedge y = 1) \vee (x = b \wedge y = 2))), \\ &\forall x \quad (p(x) \Leftrightarrow \exists y\ (r(x,y) \wedge \neg q(y))) \ . \end{aligned} \tag{4.16}$$

A prover should be capable of handling equality in order to find a Herbrand model of a completed theory. Although equality reasoning can be embedded into a prover by adding the necessary axioms for the equality predicate (i.e., axiomatic equality reasoning), it is more efficient to have dedicated equality inference rules. In axiomatic equality reasoning, equality sign is viewed as equality predicate and the equational axioms necessary for interpreting this equality predicate as an equality relation are added to the input theory. Unfortunately, Darwin does not have efficient equality reasoning yet.[4] However, Darwin has a command-line option for enabling axiomatic equality reasoning. Equality reasoning is relatively easy for finite model builders since the equality predicate has a fixed interpretation (see Section 3.2).

Herbrand semantics has a fixed meaning for equality unlike the first-order semantics [27]. Every ground term in the universe is unique. This is known as unique name assumption. Thus, extra axioms are needed in order to restrict the interpretation of the equality predicate to hold the unique name assumption. We now define the transformation which generates these axioms in order to make all the objects of a logic program unique.

Definition 4.3.2 *For a set of n objects $O = \{o_1, \cdots o_n\}$, $Una(O)$ denotes the set of inequality axioms $\{o_i \neq o_j \mid 1 \leq i \leq n-1$ and $i < j \leq n\}$ needed for the unique name assumption. For a logic program Π, we use $Una(\Pi)$ as $Una(Obj(\Pi))$ with a slight abuse of notation.*

The following is the rewrite of Fages' theorem using transformations *Comp* and *Una*.

Theorem 4.3.3 (Fages' Theorem [19]) *For a tight normal logic program Π, Herbrand models of the theory $Comp(\Pi) \cup Una(\Pi)$ coincide with its answer sets.*

Example 4.3.4 *For $\mathcal{P}$, $Una(\mathcal{P})$ denotes the set of axioms*

$$a \neq b,\ a \neq 1,\ a \neq 2,\ b \neq 1,\ b \neq 2,\ 1 \neq 2 \ . \tag{4.17}$$

[4] Although the ME calculus with equality has been defined in [11], it has not been implemented in Darwin yet.

As stated in Theorem 4.3.3, the model of $Comp(\mathcal{P}) \cup Una(\mathcal{P})$ found by Paradox (and also by FM-Darwin) coincides with the unique answer set of program $\mathcal{P}$. This model has a finite domain $\{1', 2', 3', 4'\}$ and is defined as

$1 = 1'$
$2 = 2'$
$a = 3'$
$b = 4'$

p	$1'$	$2'$	$3'$	$4'$
	F	F	T	F

q	$1'$	$2'$	$3'$	$4'$
	F	T	F	F

r	$1'$	$2'$	$3'$	$4'$
$1'$	F	F	F	F
$2'$	F	F	F	F
$3'$	T	F	F	F
$4'$	F	T	F	F

.

We have computed an answer set of a tight and consistent logic program by finite model builders Paradox and FM-Darwin. Paradox and FM-Darwin may not find the unsatisfiability of the input theory that is formed for an inconsistent program, since they are not refutationally complete in general (see Section 3.2). Later in this section, we will describe a method which will also work for inconsistent programs (i.e., finite model builders can halt and output the unsatisfiability of the formed input theory).

The first-order formulas can easily be input to Darwin, Paradox, and FM-Darwin in TPTP format.[5] However, since the search procedures of these provers work only with clauses, an input with arbitrary formulas should be clausified first. While Darwin and FM-Darwin use eprover[6] as a clausifier, Paradox can do the clausification itself if the input is in non-clausal form.

Recall that Darwin is not complete for finite satisfiability in general. It does not halt for theory $Comp(\mathcal{P}) \cup Una(\mathcal{P})$ because of the Skolem function introduced during clausification and equality reasoning. The existential variable in the last formula of (4.16) causes the introduction of a Skolem function f during clausification. The substitution axiom $f(x) = f(y) \leftarrow x = y$ is added by Darwin for equality reasoning. When an equality such as $f(a) = 1$ is in the Herbrand model of the theory, the substitution axiom endlessly generates new equations that make the Herbrand model infinite. For theory $Comp(\mathcal{P}) \cup Una(\mathcal{P})$, Darwin generates the equations $\{f(a) = 1,\ f(f(a)) = f(1),\ f(f(f(a))) = f(f(1)),\ \cdots\}$ endlessly, and cannot output a model.

[5] TPTP Syntax, http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html, visited on June 2009
[6] The E Equational Theorem Prover, http://www.eprover.org/, visited on June 2009

We propose to eliminate the Skolem functions that are introduced during clausification in order to use model generation theorem provers to compute answer sets. In this way, we can not only use Darwin to compute an answer set, but also get an input theory which is in EPR class. All the provers used in our work are deciders for EPR class of formulas. Next, we define the clausification transformation *Claus* and the completion $Comp'$ which is slightly different version of *Comp* in order to have a polynomial sized clausification using *Claus*. Then, we will define the transformation *ElimSkol* that eliminates the Skolem functions.

Definition 4.3.5 (*Comp'*) *The completion $Comp'(h)$ of a predicate h of a logic program Π is the same transformation as $Comp(h)$ unless h is defined by at least two rules which have more than one conjuncts. In that case, $Comp'(h)$ denotes the set of formulas*

$$\forall \overrightarrow{u} \ \Big(h(\overrightarrow{u}) \ \Leftrightarrow \ h\_aux_1(\overrightarrow{u}) \vee \cdots \vee h\_aux_k(\overrightarrow{u})\Big) \tag{4.18}$$

$$\forall \overrightarrow{u} \ \Big(h\_aux_i(\overrightarrow{u}) \ \Leftrightarrow \ ConjPart(r_i)\Big) \quad \textit{for all } 1 \leq i \leq k \ , \tag{4.19}$$

where $r_1, \cdots, r_k$ are the rules that have h in the head and $h\_aux_1, \cdots, h\_aux_k$ are the auxiliary predicates that are introduced for every generating rule and have the same arity as h. Note that $k \geq 2$ and at least two $ConjPart(r_i)$s have more than one conjunct for the case where Comp and $Comp'$ differ.

Example 4.3.6 *Considering the program $\mathcal{P}$ in Example 4.3.1, $Comp'(\mathcal{P})$ denotes the set of formulas;*

$$\begin{aligned}
&\forall x \quad (q(x) \Leftrightarrow (x = 2)),\\
&\forall xy \quad (r(x,y) \Leftrightarrow (r_aux_1(x,y) \vee r_aux_2(x,y))),\\
&\forall xy \quad (r_aux_1(x,y) \Leftrightarrow (x = a \wedge y = 1)),\\
&\forall xy \quad (r_aux_2(x,y) \Leftrightarrow (x = b \wedge y = 2)),\\
&\forall x \quad (p(x) \Leftrightarrow \exists y \ (r(x,y) \wedge \neg q(y))) \ .
\end{aligned} \tag{4.20}$$

Note that, the completions of the predicates $Comp'(p)$ and $Comp'(q)$ in (4.20) are equal to the ones in (4.16) since both p and q have only one generating rule.

The naive procedure that converts any first-order sentence to one in implicative or conjunctive normal form may generate an exponential number of clauses [45, 51]. This is the case when the completion formula of a predicate has at least two *ConjPart*s that include more than one

literal. In such cases $Comp'$ introduces auxiliary predicates for each $ConjPart$ in order to avoid exponential sized clausification. The naive procedure applied to $Comp'$ will produce a polynomial sized clausification. $Claus(Comp'(\Pi))$, which will be described later in this section, denotes the set of clauses produced by this type of clausification. Similarly, the definitional approach in clausification tries to reduce the number of clauses by introducing definitions [51]. Moreover, Cmodels introduces new propositional symbols for the rule bodies for polynomial sized clausification [5].

Following is the theorem that shows the correctness of the transformation $Comp'$.

Theorem 4.3.7 *For a normal logic program* Π*,* M' *is a Herbrand model of* $Comp'(\Pi)$ *iff* M *is a Herbrand model of* $Comp(\Pi)$ *such that* M' *and* M *coincide on predicates* $Pred(\Pi)$*.*

Proof. ($\Leftarrow$-part) Suppose that M is a Herbrand model of $Comp(\Pi)$. We will construct M' from M and prove that it is a Herbrand model of $Comp'(\Pi)$. Initially, set $M' = M$. Note that for a predicate t, for which $Comp'(t)$ does not introduce any auxiliary predicates, $Comp'(t) = Comp(t)$. Let h be a predicate of arity n which has auxiliary predicates in completion $Comp'(h)$. Let $r_1, \cdots, r_k$ be the rules in Π which generate h. $Comp(h)$ is of the form (4.3). Since M satisfies $Comp(h)$, for every instance of h in M, there must be at least one $ConjPart(r_j)$ for $1 \leq j \leq k$, whose corresponding instance is satisfied by M, and vice versa (i.e., for every instance of the subformula $ConjPart(r_j)$ for $1 \leq j \leq k$ satisfied by M, there must be corresponding instance of h in M). For every instance $h(\overrightarrow{b}) \in M$ where $\overrightarrow{b}$ is a tuple of n ground terms, add $h\_aux_j(\overrightarrow{b})$ to M' if M satisfies $ConjPart(r_j)\theta$ for any $1 \leq j \leq k$ where $\theta = \{u_1/b_1, \cdots, u_n/b_n\}$. We repeat this augmentation for every predicate for which $Comp'$ introduces an auxiliary predicate. It is clear that M' and M coincides on predicates $Pred(\Pi)$ since we add instances of auxiliary predicates only. Now, we have to show that M' satisfies $Comp'(h)$ (i.e., formulas (4.18) and (4.19)) in order to be a model of $Comp'(\Pi)$. Take any ground instance of formula (4.18). Suppose that $h(\overrightarrow{t})$ is in the left hand side of the bi-conditional connective of the selected instance. If $h(\overrightarrow{t}) \in M'$, then there exists some r_j for $1 \leq j \leq k$ such that an instance of $ConjPart(r_j)$ is satisfied by M' (note that $M \subseteq M'$ and M satisfies (4.3)). By the construction of M', $h\_aux_j(\overrightarrow{t}) \in M'$. Hence, M' satisfies one half of the logical equivalence that is represented by the ground instance of (4.18). For the other half, suppose that $h\_aux_j(\overrightarrow{t})$ for $1 \leq j \leq k$ is in M'. From the construction of M', there should be a corresponding instance of $ConjPart(r_j)$ satisfied by M (M' also). Since M satisfies (4.3)

and $M \subseteq M'$, $h(\overrightarrow{t}) \in M'$. Thus, M' satisfies any instance of (4.18). It is more straightforward to see that M' satisfies the formulas (4.19). In the construction of M', we add instances of auxiliary predicates when corresponding instances of $ConjPart(r_j)$ for any $1 \leq j \leq k$ are satisfied and this is the only way we add them.

($\Rightarrow$-part) Suppose that M' is a Herbrand model of $Comp'(\Pi)$. In order to construct M, eliminate all instances of auxiliary predicates in M' that are introduced by $Comp'$. We want to prove that M is a model of $Comp(\Pi)$ by showing that M satisfies any instance of the formula (4.3) where h is such a predicate described in the $\Leftarrow$-part of the proof (i.e., any predicate for which $Comp'$ introduces auxiliary predicates). For any instance $ConjPart(r_j)\theta$ for $1 \leq j \leq k$ and $\theta = \{u_1/b_1, \cdots, u_n/b_n\}$, if it is satisfied by M', $h\_aux_j(b_1, \cdots, b_n)$ is in M' since M' satisfies formulas (4.19). Moreover, there is the corresponding instance $h(b_1, \cdots, b_n)$ in M' since it satisfies the formula (4.18). M also satisfies the subformulas $ConjPart(r_j)\theta$ and $h(b_1, \cdots, b_n)$ since these subformulas have predicates only from $Pred(\Pi)$ and in the construction of M instances of only auxiliary predicates are eliminated. Hence, M satisfies one half of the bi-conditional connective in formula (4.3). Similarly, for any instance $h(b_1, \cdots, b_n)$ in M', there is at least one j for $1 \leq j \leq k$ where $h\_aux_j(b_1, \cdots, b_n)$ is in M' because of the formula (4.18) and moreover, M' satisfies $ConjPart(r_j)\theta$ because of formulas (4.19). From the construction of M, we know that both $h(b_1, \cdots, b_n)$ and $ConjPart(r_j)\theta$ are also satisfied by M.Hence, M satisfies the other half of the bi-conditional connective in formula (4.3). ■

Definition 4.3.8 (*Claus*) *Claus is the naive clausification procedure that converts any first-order sentence to one in implicative or conjunctive normal form.*[7] *Details of the procedure can be found in any textbook in logic (see eg. [45]).*

As stated in the definition of $Comp'$, we use $Claus(Comp'(\Pi))$ instead of $Claus(Comp(\Pi))$ in order to keep the number of clauses polynomial. Let h be a predicate where $Comp'(h)$ is the set of formulas (4.18) and (4.19). *Claus* will generate the following clauses related with

[7] In [46], we had used a slightly different clausification transformation, which we could not observe enough improvement in terms of runtime. Thus, we used native clausification in this work.

h.

$$\neg h(\overrightarrow{u}) \vee h\_aux_1(\overrightarrow{u}) \vee \cdots \vee h\_aux_k(\overrightarrow{u}) \tag{4.21}$$

$$\left.\begin{array}{l} \neg h\_aux_1(\overrightarrow{u}) \vee h(\overrightarrow{u}) \\ \vdots \\ \neg h\_aux_k(\overrightarrow{u}) \vee h(\overrightarrow{u}) \\ h\_aux_i(\overrightarrow{u}) \vee \neg c_{i1} \vee \cdots \vee \neg c_{ij} \\ \\ \neg h\_aux_i(\overrightarrow{u}) \vee Skol(c_{i1}) \\ \vdots \\ \neg h\_aux_i(\overrightarrow{u}) \vee Skol(c_{ij}) \end{array}\right\} \; 1 \leq i \leq k \tag{4.22}$$

The native clausification *Claus* generates clauses (4.21) for the formula (4.18) and (4.22) for the set of formulas (4.19) where $ConjPart(r_i) = \exists\overrightarrow{y}(c_{i1} \wedge \cdots \wedge c_{ij})$ and $Skol(c)$ denotes the Skolemization [45] of a literal c which is in $ConjPart$ of a completed formula of the form (4.18). For instance, let $ConjPart(r) = \exists x(t(x, u_1) \wedge s(u_2))$ where u_1 and u_2 are all the universally quantified variables in the corresponding completed formula and x is a existentially quantified variable. For *Claus*, $Skol(t(x, u_1)) = t(f(u_1, u_2), u_1)$ and $Skol(s(u_2)) = s(u_2)$. Note that $\overrightarrow{y}$ can be empty list of variables when there are no existential variables introduced during the completion.

Example 4.3.9 *Considering the program $\mathcal{P}$ in Example 4.3.1, Claus($Comp'(\mathcal{P})$) denotes the set of clauses;*

$$\begin{array}{ll} & r\_aux_1(a, 1), \\ & r\_aux_2(b, 2), \\ q(2), & x = a \vee \neg r\_aux_1(x, y), \\ x = 2 \vee \neg q(x), & y = 1 \vee \neg r\_aux_1(x, y), \\ & x = b \vee \neg r\_aux_2(x, y), \\ p(x) \vee \neg r(x, y) \vee q(y), & y = 2 \vee \neg r\_aux_2(x, y), \\ r(x, f(x)) \vee \neg p(x), & \\ \neg q(f(x)) \vee \neg p(x), & r\_aux_1(x, y) \vee r\_aux_2(x, y) \vee \neg r(x, y), \\ & r(x, y) \vee \neg r\_aux_1(x, y), \\ & r(x, y) \vee \neg r\_aux_2(x, y) \;\; . \end{array} \tag{4.23}$$

Some trivial simplifications are done; for instance, the clause $q(x) \vee x \neq 2$ is simplified to the unit clause $q(2)$. The existential variable in the last formula of (4.20) causes the introduction of a Skolem function f in (4.23) during clausification.

Using the correctness result of $Comp'$ (Theorem 4.3.7), we can replace $Comp(\Pi)$ in Theorem 4.3.3 with $Comp'(\Pi)$, and then apply the clausification *Claus*. Note that all the quantifiers range over the Herbrand universe of the input in Herbrand logic [27] (i.e., the domain is closed when we are searching for Herbrand models). The domain (i.e., Herbrand universe) of the theory formed in Theorem 4.3.3 is the set $Obj(\Pi)$ of objects of the program Π. However, *Claus* can extend the domain of the input with new Skolem functions. Thus, in order to compute answer sets using a theory formed by *Claus*, we should add the domain closure axiom [44] to the clausified theory to force that these Skolem functions do not denote new objects. For some class of programs, the addition of the domain closure axiom is not necessary. This will be discussed later in Section 4.6 in detail. The following is the definition of domain closure axiom for a logic program and Theorem 4.3.11 is a rewrite of the Fages' theorem (Theorem 4.3.3) using transformations *Claus*, $Comp'$, *Una* and *Dca*.

Definition 4.3.10 (*Dca*) *For a logic program* Π*,* $Dca(\Pi)$ *denotes the domain closure axiom*

$$\forall x(x = o_1 \vee \cdots \vee x = o_r) \ , \tag{4.24}$$

where $Obj(\Pi) = \{o_1 \cdots o_r\}$.

Theorem 4.3.11 (Fages' Theorem using *Claus*) *For a tight normal logic program* Π*, Herbrand models of the theory* $Claus(Comp'(\Pi)) \cup Una(\Pi) \cup Dca(\Pi)$ *coincide with its answer sets.*

Proof. It basically follows from Theorem 4.3.3 and Theorem 4.3.7. We need to show that there exists a Herbrand model of $Comp(\Pi) \cup Una(\Pi)$ iff $Claus(Comp'(\Pi)) \cup Una(\Pi) \cup Dca(\Pi)$ has a corresponding Herbrand model. The proof is straightforward. However, it may not be obvious to see the role of the domain closure axiom $Dca(\Pi)$. When $Comp(\Pi) \cup Una(\Pi)$ has no models (i.e., Π has no answer sets), there can be a model of $Claus(Comp'(\Pi)) \cup Una(\Pi)$, i.e., when we do not add the domain closure axiom. Example 4.6.1 in Section 4.6 displays this situation more clearly. ■

Next, we will define the transformation *ElimSkol* which takes $Claus(Comp'(\Pi))$ as input and eliminates the Skolem functions in it by introducing new predicates. Similar techniques of eliminating functions by introducing predicates are applied in [7, 13].

Definition 4.3.12 (*ElimSkol*) *ElimSkol does not change a clause unless it has a Skolem function. For any clause with a Skolem function in the given set of clauses, ElimSkol performs the transformation*

$$C' \vee (\neg)P(\cdots, f(a_1, \cdots, a_n), \cdots) \rightsquigarrow C' \vee (\neg)P(\cdots, x, \cdots) \vee \neg R_f(a_1, \cdots, a_n, x) \quad (4.25)$$

where P is the literal that has the eliminated Skolem function f, C' is the disjunction of other literals in the clause, $a_1, \cdots, a_n$ are terms, x is a fresh variable and R_f is the newly introduced predicate. Furthermore, for each introduced predicate, ElimSkol adds the following axioms

$$R_f(x_1, \cdots, x_n, o_1) \vee \cdots \vee R_f(x_1, \cdots, x_n, o_k) \quad (4.26)$$

$$y = y' \vee \neg R_f(x_1, \cdots, x_n, y) \vee \neg R_f(x_1, \cdots, x_n, y') \quad (4.27)$$

where $Obj(\Pi) = \{o_1, \cdots, o_k\}$ and $x_1, \cdots, x_n, y, y'$ are variables.

An n-ary Skolem function can be eliminated by introducing an $n + 1$-ary predicate. Intuitively, the last argument of R_f in (4.25) denotes the value of the Skolem function f applied to the first n arguments. Hence, the relation for the interpretation of R_f in a model of $ElimSkol(Claus(Comp'(\Pi)))$ should define the Skolem function f. This necessitates that the interpretation of R_f will be left-total over the set $Obj(\Pi)$ (i.e., for every $o_1, \cdots, o_n \in Obj(\Pi)$ there exists $b \in Obj(\Pi)$ such that $\langle o_1, \cdots, o_n, b\rangle$ is included in the relation) and be right-unique (i.e., if $\langle o_1, \cdots, o_n, b\rangle$ is in the relation, then there is no element of the form $\langle o_1, \cdots, o_n, b'\rangle$ in the relation such that $b \neq b'$).[8] Thus, *ElimSkol* adds the totality (4.26) and the uniqueness (4.27) axioms to the input set of clauses.

Note that the predicate that has the Skolem function as an argument can also be an equality predicate. In that case, the transformation (4.25) can be seen as

$$C' \vee f(a_1, \cdots, a_n)(\neq) = y \rightsquigarrow C' \vee x(\neq) = y \vee \neg R_f(a_1, \cdots, a_n, x) \ . \quad (4.28)$$

[8] Functions are left-total and right-unique relations [7].

Example 4.3.13 *Considering $\mathcal{P}$ in Example 4.3.1, ElimSkol(Claus(Comp′($\mathcal{P}$))) denotes the set of clauses*

$$
\begin{array}{ll}
 & r\_aux_1(a,1),\\
 & r\_aux_2(b,2),\\
q(2), & x = a \vee \neg r\_aux_1(x,y),\\
x = 2 \vee \neg q(x), & y = 1 \vee \neg r\_aux_1(x,y),\\
 & x = b \vee \neg r\_aux_2(x,y),\\
p(x) \vee \neg r(x,y) \vee q(y), & y = 2 \vee \neg r\_aux_2(x,y),\\
r(x,y) \vee \neg p(x) \vee \neg R_f(x,y), & \\
\neg q(y) \vee \neg p(x) \vee \neg R_f(x,y), & r\_aux_1(x,y) \vee r\_aux_2(x,y) \vee \neg r(x,y),\\
 & r(x,y) \vee \neg r\_aux_1(x,y),\\
 & r(x,y) \vee \neg r\_aux_2(x,y),
\end{array}
\tag{4.29}
$$

$$
\begin{array}{l}
R_f(x,a) \vee R_f(x,b) \vee R_f(x,1) \vee R_f(x,2),\\
y = y' \vee \neg R_f(x,y) \vee \neg R_f(x,y') \ .
\end{array}
$$

The Skolem function f in (4.23) is eliminated by introducing the binary predicate R_f in (4.29). The last two clauses are the totality and uniqueness axioms for R_f.

Following is the theorem that shows the correctness of the transformation *ElimSkol*. This theorem additionally shows that there is no need to add the domain closure axiom to the theory $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$ since the totality axioms constrain the Skolem functions in the same way the domain closure axiom does. Later, Corollary 4.3.16 states this result with regard to answer set computation.

Theorem 4.3.14 *For a normal logic program Π, M is a Herbrand model of the theory $Claus(Comp'(\Pi)) \cup Dca(\Pi) \cup Una(\Pi)$ iff M' is a Herbrand model of the theory $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$, such that M' and M coincide on predicates in $Pred(\Pi)$.*

Proof. ($\Rightarrow$-part) Suppose that M is a Herbrand model of $Claus(Comp'(\Pi)) \cup Dca(\Pi) \cup Una(\Pi)$. We will construct M', a Herbrand model of $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$, from M. M' will include every atom in M, however any Skolem function f found in an atom is substituted with its value which is defined by the interpretation of equality predicate in M.

Note that equality is handled as a predicate and the Herbrand model has equality atoms. Let σ be a substitution $\{f(a_1, \cdots, a_n) \mapsto d \mid f$ is any Skolem function introduced by $Claus$ and $f(a_1, \cdots, a_n) = d \in M\}$. Formally, the initial $M' = \{L\sigma | L \in M\}$. Next, for every Skolem function f, we will add R_f atoms to M'. For every equality $f(a_1, \cdots, a_n) = d$ in M, add $R_f(a_1, \cdots, a_n, d)$ atom to M'. M and M' coincide on $Pred(\Pi)$ since they both assign each propositional atom of Π same value.

Next, we will prove that M' is a model of $ElimSkol(Claus(Comp'(\Pi)))$. $ElimSkol$ does not change clauses without Skolem functions. Such clauses in $ElimSkol(Claus(Comp'(\Pi)))$ are satisfied by M' trivially since M' includes all atoms from M which have no occurrence of a Skolem function. Any clause of $Claus(Comp'(\Pi))$ which has a Skolem function is of the form,

$$C' \vee (\neg)P(\cdots, f(x_1, \cdots, x_n), \cdots) \tag{4.30}$$

where P is the literal that has a Skolem function f and C' is the disjunction of other literals in the clause. We assume that C' does not contain Skolem functions. $ElimSkol$ transforms (4.30) to

$$C' \vee (\neg)P(\cdots, x, \cdots) \vee \neg R_f(x_1, \cdots, x_n, x) \tag{4.31}$$

as defined in Section 4.3. Take any instance of (4.31),

$$C'\theta \vee (\neg)P(\cdots, d, \cdots) \vee \neg R_f(a_1, \cdots, a_n, d) \tag{4.32}$$

where θ includes the substitutions $\{x_1 \mapsto a_1, \cdots, x_n \mapsto a_n, x \mapsto d\}$. If $R_f(a_1, \cdots, a_n, d) \notin M'$ then (4.32) is satisfied by M' trivially. If $R_f(a_1, \cdots, a_n, d) \in M'$, it follows from the construction of M' that $f(a_1, \cdots, a_n) = d \in M$. The following clause is the corresponding instance of (4.30) in $Claus(Comp'(\Pi))$.

$$C'\theta \vee (\neg)P(\cdots, f(a_1, \cdots, a_n), \cdots) \tag{4.33}$$

- If $L \in M$ is an atom in $C'\theta$ and satisfies (4.33), $L\sigma \in M'$ will satisfy (4.32) since L has no Skolem function and $L\sigma = L$.

- If $P(\cdots, f(a_1, \cdots, a_n), \cdots) \in M$ is the atom satisfying (4.33), then $P(\cdots, d, \cdots) = P(\cdots, f(a_1, \cdots, a_n), \cdots)\sigma \in M'$ satisfies (4.32). Note that since $f(a_1, \cdots, a_n) = d \in M$, σ includes a substitution $f(a_1, \cdots, a_n) \mapsto d$.

Thus, M' satisfies any instance of (4.31). For the validity of this result, the assumption made above is not needed, but it makes the proof much clearer. If C' has a literal that has a Skolem function, one can designate it as in the place of P in (4.30).

Since M satisfies $Dca(\Pi)$ and $Una(\Pi)$ the equalities related with any Skolem function f will define a function which is left-total and right-unique over the set $Obj(\Pi)$. Thus, R_f predicates that are added to M' will satisfy the totality axiom (4.26) and the uniqueness axiom (4.27). The selection of the Skolem function f is arbitrary. Hence, M' will satisfy all totality and uniqueness axioms defined for all the introduced predicates.

($\Leftarrow$-part) Let M' be a Herbrand model of $ElimSkol(Claus(Comp'(\Pi)))$. Initially, we set $M = M' \setminus R$ where R is the set of instances of predicates that are introduced for Skolem functions (i.e., atoms of R_f predicate for every Skolem function f). Next, we will define f using R_f atoms in M'. Define $D_f = \{f(a_1, \cdots, a_n) = d \mid R_f(a_1, \cdots, a_n, d) \in M', a_1, \cdots, a_n, d \in Obj(\Pi)\}$. Add D_f to M for every Skolem function f. Some of the atoms in M are applicable for substitutions of terms using the equalities of Skolem functions that are newly defined. For instance, $p(\cdots, x, \cdots) \vee \neg p(\cdots, y, \cdots) \vee x \neq y$ is a substitution axiom of a predicate p defined for equality reasoning. If $p(\cdots, d, \cdots) \in M$, add $p(\cdots, f(a_1, \cdots, a_n), \cdots)$ to M using the substitution axiom and the equality $f(a_1, \cdots, a_n) = d$ added to M. After performing all possible substitutions, we get the final M that is a model of $Claus(Comp'(\Pi)) \cup Dca(\Pi) \cup Una(\Pi)$. Any clause which does not include a Skolem function in $ElimSkol(Claus(Comp'(\Pi)))$ is also in $Claus(Comp'(\Pi))$. M will satisfy them trivially since $M \supseteq M' \setminus R$. Any clause of $Claus(Comp'(\Pi))$ which has a Skolem function is transformed by $ElimSkol$ to a form (4.31) where P is the literal of interest that originally has a Skolem function f and R_f is the predicate introduced for f. As in the $\Rightarrow$-part, we assume that C' does not contain a Skolem function for the clarity of the proof. The corresponding clause in $Claus(Comp'(\Pi))$ is of the form (4.30). Take any instance of (4.30),

$$C'\omega \vee (\neg)P(\cdots, f(a_1, \cdots, a_n), \cdots) \tag{4.34}$$

where ω includes the substitutions $\{x_1 \mapsto a_1, \cdots, x_n \mapsto a_n\}$. The corresponding instance of (4.31) in $ElimSkol(Claus(Comp'(\Pi)))$ is of the following form.

$$C'\omega \vee (\neg)P(\cdots, x, \cdots) \vee \neg R_f(a_1, \cdots, a_n, x) \tag{4.35}$$

Since M' satisfies the totality and uniqueness axioms for R_f, there should be one atom $R_f(a_1, \cdots, a_n, d) \in M'$. It follows from the construction of M that $f(a_1, \cdots, a_n) = d \in M$.

Except the following instance of (4.35),

$$C'\omega \vee (\neg)P(\cdots, d, \cdots) \vee \neg R_f(a_1, \cdots, a_n, d) \tag{4.36}$$

M' satisfies all instances of (4.35) trivially.

- If $L \in M'$ is an atom in $C'\omega$ and satisfies (4.36), $L \in M$ satisfies (4.34) since L has no Skolem function and $M \supseteq M' \setminus R$.
- If $P(\cdots, d, \cdots) \in M'$ is the atom satisfying (4.36), $P(\cdots, f(a_1, \cdots, a_n), \cdots) \in M$ satisfies (4.34). Note that $P(\cdots, f(a_1, \cdots, a_n), \cdots)$ is in M because of the substitution performed with $f(a_1, \cdots, a_n) = d \in M$ and $P(\cdots, d, \cdots) \in M$ during the construction of M.

Thus, M satisfies any instance of (4.30).

Note that the set of objects of theory $ElimSkol(Claus(Comp'(\Pi)))$ is $Obj(\Pi)$ and the difference from that of $Claus(Comp'(\Pi))$ is the eliminated Skolem functions. Take any Skolem function f. Since a function can be defined from the interpretation of R_f in M',[9] we have an object for every possible term $f(a_1, \cdots, a_n)$. Additionally, since M' satisfies the totality axiom (4.26), it is guaranteed that all equalities having f is of the form $f(a_1, \cdots, a_n) = o_i$ where $o_i \in Obj(\Pi)$. These equalities satisfy all instances of the domain closure axiom defined for f. From arbitrary selection of the Skolem function, we conclude that M satisfies $Dca(\Pi)$. ■

We can replace $Claus(Comp'(\Pi))$ with $ElimSkol(Claus(Comp'(\Pi)))$ in Theorem 4.3.11 using Theorem 4.3.14. The following is the rewrite of Fages' theorem using the transformations $ElimSkol$, $Claus$, $Comp'$ and Una.

Theorem 4.3.15 (Fages' Theorem using *ElimSkol*) *For a tight normal logic program* Π*, Herbrand models of the theory* $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$ *coincide with its answer sets.*

Proof. It follows from Theorem 4.3.11 and Theorem 4.3.14. ■

The following corollary is an interesting result of Theorem 4.3.14 and Theorem 4.3.15 about the need of domain closure axiom for computing answer sets.

[9] Since R_f is a left-total and right-unique relation, one can define a function f for it [7].

Corollary 4.3.16 *When computing an answer set of a tight normal logic program* Π *using the transformation ElimSkol, there is no need for the addition of domain closure axiom* $Dca(\Pi)$ *to the theory (i.e.,* $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$*).*

As an application of Theorem 4.3.15, Darwin can find a model for the theory (4.29) expanded with $Una(\mathcal{P})$ which corresponds to the unique answer set of the program $\mathcal{P}$. The model, represented as a disjunction of implicit generalizations (DIG[10]) [20], is

$$\begin{array}{ll} x = x & q(2) \\ R_f(a,1) & r(a,1) \\ R_f(x,2) & r(b,2) \\ \quad \text{with exception: } R_f(a,2) & r\_aux_1(a,1) \\ p(a) & r\_aux_2(b,2) \end{array} .$$

For the theory $ElimSkol(Claus(Comp'(\mathcal{P}))) \cup Una(\mathcal{P})$, both Paradox and FM-Darwin can find a model which corresponds to the unique answer set. In general, given a normal logic program the theory generated by the transformation *ElimSkol* is in EPR class. Hence, both provers are deciders for this type of input, just like Darwin.

4.4 Types of Input Generated by Transformations

Below is the summary of the application of various transformations for computing an answer set of a tight logic program Π using model generation theorem provers. Actually, the experiments in Section 6 use these three types of input theories. The completeness results of the model generation theorem provers for these input types are also examined.

- $Comp(\Pi) \cup Una(\Pi)$ gives non-clausal input. If Π is consistent, finite model builders can find a model of this input that corresponds to an answer set since they are complete for finite satisfiability. Due to Skolem functions introduced in the clausification process performed by the provers, the Herbrand universe of the input can be infinite. This may cause Darwin to perform an infinite derivation and not to halt. However, if Π has no answer sets, Darwin will find the unsatisfiability of this input since it is refutation-

[10] Basically, a model is represented by positive atoms with exceptions of negative atoms.

ally complete. This time, Paradox and FM-Darwin may not halt because they are not refutationally complete in general.

- $Claus(Comp'(\Pi)) \cup Una(\Pi)$ gives clausal input which may contain Skolem functions. For this input, the provers do not need to clausify it first. We used clausal input with Skolem functions in the experiments just to compare the clausification performed by the provers and that performed by the procedure *Claus* which is described in this section. The expected behaviors of the provers for this type of input are exactly the same as those mentioned in the previous item.

- $ElimSkol(Claus(Comp'(\Pi))) \cup Una(\Pi)$ gives clausal input with no Skolem functions. This type of input is in Bernays-Schönfinkel class. All the provers used in this work are deciders for this class of input. If Π is consistent, all the provers will output a model which corresponds to an answer set of Π; otherwise, they will find the unsatisfiability of the input when there are enough time and space resources.

Although the first two types of input need domain closure axiom in general, we have not added it to inputs because of the special class of input logic programs that are used in experiments. Domain-restricted programs, which are quite general in answer set programming, do not need the addition of the domain closure axiom. For details of this result see Section 4.6.

4.5 Generating an Input Theory Suitable for Sort Optimization

The finite model builders Paradox and FM-Darwin use sort optimization in order to reduce the number of domain elements needed for forming a finite model (see Section 3.2). For instance, the logic program $\mathcal{P}$ is suitable for a multi-sorted theory. While the constants a and b can be of one sort, 1 and 2 can be of another. A finite model builder can find a model with a domain of size 2 instead of 4 if it utilizes sort optimization. Equal domain elements can be used to interpret the constants $\{a, b\}$ and $\{1, 2\}$.

Paradox and FM-Darwin infer sort information from the input theory [13]. The sort detection algorithm uses the fact that same variables in a clause and arguments of equality predicates should be of the same sort. However, all the inputs mentioned so far do not allow the provers to infer more than one sort. The inequalities in the unique name assumption axioms generated

by *Una* constrain all the constants of the input to be in the same sort since finite model builders infer that arguments of the equality symbol should be of the same sort.

In order to utilize sort optimization, Completor infers sorts from a logic program. The sort inference algorithm of Completor is based on the one in [13]. Here, Completor uses the fact that same variables in a rule and arguments of equality predicates should be of the same sort. After inferring sorts, it generates the unique name assumption axioms according to the sort information, i.e., unique name assumption axioms for each sort independently. For instance, the unique name assumption axioms for $\mathcal{P}$ should be $\{a \neq b, 1 \neq 2\}$ instead of (4.17). Also, the totality axioms generated by *ElimSkol* should also respect the sort information. In (4.29), the totality axiom should be $R_f(x, 1) \vee R_f(x, 2)$ since the sort of the eliminated Skolem function is $\{1, 2\}$.

If we give *Comp*(Π) and the unique name assumption axioms respecting the sorts to Paradox as input, it will find the following model with a domain of size 2, which corresponds to the unique answer set.

$$1 = 1' \quad 2 = 2' \quad a = 1' \quad b = 2'$$

p	$1'$	$2'$
	T	F

q	$1'$	$2'$
	F	T

r	$1'$	$2'$
$1'$	T	F
$2'$	F	T

The logic program encoding of the blocks world problem in Section 6.4 is suitable for sort optimization. The runtime results in that section demonstrate the advantage of utilizing sort optimization in computing an answer set.

4.6 No Need for the Domain Closure Axiom for Domain-restricted Programs

When we are computing an answer set of a tight normal logic program using the transformation *Claus*, we need to add the domain closure axiom to the input theory as stated in Theorem 4.3.11. Furthermore, it is also needed for the input theory generated by *Comp*. Actually, the non-clausal input formed by *Comp* does not seem to need the domain closure axiom as stated in Theorem 4.3.3. However, all the theorem provers used work on clauses and have internal clausifiers which clausify the input first. This aspect of provers necessitate the addition of

domain closure axiom. Following is an interesting logic program which shows this need in order to compute an answer set in a sound manner using model generation theorem provers.

Example 4.6.1 *Let logic program $\mathcal{N}$ be*

$$\begin{aligned} & p(a). \\ & k \leftarrow not\ p(X). \\ & \leftarrow not\ k. \end{aligned} \tag{4.37}$$

$\mathcal{N}$ is tight and has no answer sets. The non-clausal theory $Comp(\mathcal{N})$ is

$$\begin{aligned} & \forall x\ (p(x) \Leftrightarrow (x = a)), \\ & k \Leftrightarrow \exists y\ (\neg p(y)), \\ & \bot \Leftarrow \neg k\ . \end{aligned} \tag{4.38}$$

The theory $Comp(\mathcal{N})$ has no Herbrand models as expected.[11] However, $Claus(Comp'(\mathcal{N}))$ has a model. For instance, the set $\{k,\ p(1'),\ \neg p(2')\}$ is a satisfying model with domain $\{1', 2'\}$. The model generation theorem provers find this model for the input $Claus(Comp'(\mathcal{N}))$. Additionally, they also output the same model for the input $Comp(\mathcal{N})$, since they clausify it first. Note that, altough this model is a non-Herbrand model of $Comp(\mathcal{N})$, it is a Herbrand model of $Claus(Comp'(\mathcal{N}))$. The domain of the Skolem function introduced for the existentially quantified variable in (4.38) is not restricted to $\{a\}$, which is the set of all constants in program $\mathcal{N}$. In order to force the provers to search for Herbrand models of $Comp(\mathcal{N})$, we should add the domain closure axiom to both theories. For $\mathcal{N}$, the domain closure axiom $Dca(\mathcal{N}) = \{\forall x\ (x = a)\}$. The provers will find the unsatisfiability of the theory $Comp(\mathcal{N}) \cup Dca(\mathcal{N})$ or $Claus(Comp'(\mathcal{N})) \cup Dca(\mathcal{N})$ as expected.

There is no need to add a domain closure axiom to a clausal theory generated by *ElimSkol* as stated in Corollary 4.3.16. The totality axioms added for the introduced predicates actually restrict the domains of the eliminated Skolem functions to the set of all constants appearing in the program. For instance, the provers will find the unsatisfiability of the theory $ElimSkol(Claus(Comp'(\mathcal{N})))$ as expected.

Lparse[12], which is a grounder, requires that the input logic program is *domain-restricted* [52].[13] All of the rules of a domain-restricted program are domain-restricted (i.e., every vari-

[11] Since there is only one object in $\mathcal{N}$, $Una(\mathcal{N}) = \{\}$.
[12] Computing the stable model semantics, http://www.tcs.hut.fi/Software/smodels/, visited on June 2009
[13] See Section 4.5 of Lparse manual http://www.tcs.hut.fi/Software/smodels/lparse.ps, visited on June 2009

able appearing in a rule also appears in a positive domain predicate in the body). Lparse uses domain predicates to restrict the domains of variables. Domain predicates can be defined by other domain predicates in a non-recursive way. Formally, domain predicates are defined by the maximal stratifiable subset of the program. Every variable in a rule of domain-restricted program must also occur in a positive body literal which is in a lower stratum than that of the head literal (for details, see [52]). The logic program in Example 4.3.1 and all the benchmark programs in Chapter 6 are domain-restricted. Note that program $\mathcal{N}$ is not domain-restricted, because the second rule has a variable which does not appear in a positive domain predicate.

Since the domain of every variable in a domain-restricted program is restricted to a subset of all constants appearing in the program, the need for adding the domain closure axiom to the theory formed by completing such a program is eliminated. If a logic program L is domain-restricted, the theories $Comp(L)$ and $Claus(Comp'(L))$ satisfy the domain closure axiom implicitly. Theorem 4.6.3 states this result formally. The following proposition is used in this theorem's proof.

Proposition 4.6.2 *From a domain-restricted normal logic program* Π*, construct* Π' *by adding the fact* $X(c_x)$ *to* Π*, where* X *and* c_x *are fresh predicate and object symbols respectively. A is an answer set of* Π *iff* A' *is an answer set* Π' *such that* $A' = A \cup \{X(c_x)\}$.

Proof. The proof uses splitting set theorem for logic programs under answer set semantics [36]. A splitting set for a program P is a set of atoms U such that for every rule r of P, if $Head(r) \in U$ then $Body^+(r) \cup Body^-(r) \subseteq U$. The splitting set splits the program into two parts. The bottom of P relative to U is denoted by $b_U(P)$. $b_U(P) = \{\ r \mid r \in P$ and $Head(r) \cup Body^+(r) \cup Body^-(r) \subseteq U\}$. The top of P is the rest of the program denoted by $P \setminus b_U(P)$. The splitting theorem says that $X \cup Y$ is an answer set of P iff X is an answer set of $b_U(P)$, Y is the answer set of $e_U(P \setminus b_U(P), X)$ and $X \cup Y$ is consistent. $e_U(P \setminus b_U(P), X)$ denotes the evaluation of the top program relative to X formed by first removing all rules that have a literal from U but not satisfied by X and then removing all the body literals which are included in U from remaining rules.

($\Rightarrow$-part) Suppose that A is an answer set of Π. Then, A is an answer set of $Ground(\Pi)$. Since X and c_x are new, Π and Π' are domain-restricted programs, $Ground(\Pi') = Ground(\Pi) \cup \{X(c_x).\}$ (from instantiation of domain-restricted programs in [52]). We can apply splitting

set theorem to $Ground(\Pi')$. Take $U = \{X(c_x)\}$. It is a splitting set for $Ground(\Pi')$ trivially since there are no occurrences of the atom $X(c_x)$ in $Ground(\Pi)$. The bottom of P, $b_U(Ground(\Pi')) = \{X(c_x).\}$ and the top is $Ground(\Pi)$. $K \cup L$ is an answer set of $Ground(\Pi')$ iff K is an answer set of $b_U(Ground(\Pi'))$ and L is an answer set of $e_U(Ground(\Pi), K)$. $K = \{X(c_x)\}$. Since A is an answer set of $Ground(\Pi)$ and $e_U(Ground(\Pi), K) = Ground(\Pi)$, we can take L as A. Thus, $A' = A \cup \{X(c_x)\}$ is an answer set of $Ground(\Pi')$ and Π'.

($\Leftarrow$-part) The proof is similar to the $\Rightarrow$-part. Suppose A' is an answer set of Π' and $Ground(\Pi')$. From the $\Rightarrow$-part, we know that $Ground(\Pi') = Ground(\Pi) \cup \{X(c_x).\}$. We can apply the splitting set theorem again. Take $U = \{X(c_x)\}$ as the splitting set. It follows from the splitting set theorem A is an answer set of $Ground(\Pi)$ and Π. ■

Theorem 4.6.3 *Let* Π *be a domain-restricted tight normal logic program.* $Claus(Comp'(\Pi)) \cup Dca(\Pi) \cup Una(\Pi)$ *has a Herbrand model* M *iff* $Claus(Comp'(\Pi)) \cup Una(\Pi)$ *has a Herbrand model* M'*, such that* M *and* M' *coincides with the same answer set of* Π.

Proof. The $\Rightarrow$-part of the proof is trivial by selecting $M' = M$.

($\Leftarrow$-part) We assume that a new domain predicate and a new object are added to Π. According to the Proposition 4.6.2, adding the rule $X(c_x)$. to Π, where X and c_x are new predicate and object symbols respectively, does not affect the answer sets of Π such that all of its answer sets will include an additional atom $X(c_x)$. This addition is not necessary, but will make the proof much clearer.

Suppose that M' is a Herbrand-model of $Claus(Comp'(\Pi)) \cup Una(\Pi)$ which coincides with an answer set A of Π. If M' satisfies $Dca(\Pi)$, then it is trivially a model of $Claus(Comp'(\Pi)) \cup Dca(\Pi) \cup Una(\Pi)$. We can set $M = M'$ and conclude that M coincides with A from Theorem 4.3.11. If M' does not satisfy $Dca(\Pi)$, there should be at least one object which is different from all objects of Π. Additionally, it should be an instance of a Skolem function since all the objects of Π satisfy $Dca(\Pi)$ by an instance of the reflexivity axiom of equality. Take any $f(a_1, \cdots, a_n)$, a ground term of such a Skolem function of arity n. M' should satisfy inequalities $f(a_1, \cdots, a_n) \neq o_i$ where $o_i \in Obj(\Pi)$, since it does not satisfy $Dca(\Pi)$. Initially, set $M = M'$. Next, we will prove that the object $f(a_1, \cdots, a_n)$ does not appear in a positive atom in M'. Suppose that it is not the case and $k(\cdots, f(a_1, \cdots, a_n), \cdots)$ is the positive literal in M'. The rules that generate the predicate k are domain-restricted. Hence, there will be a

clause in $Claus(Comp'(\Pi))$

$$\neg k(\cdots, x, \cdots) \vee d_1(\cdots, x, \cdots) \tag{4.39}$$

where the variable x is in the same place as $f(a_1, \cdots, a_n)$ within the arguments of k and d_1 is a positive domain literal which belongs to a lower stratum than k. Recall that a variable in a rule of domain-restricted program must occur in a positive domain literal that belongs to a lower startum than the head [52]. If d_1 is not in the lowest stratum, it will depend on domain predicate which is in a lower stratum. Hence, there will also be a clause

$$\neg d_1(\cdots, x, \cdots) \vee d_2(\cdots, x, \cdots) \tag{4.40}$$

which is similar to (4.39). Eventually, the variable x will be related to a domain predicate d_l, which is in the lowest stratum. Since d_l is in the lowest stratum, the extensions of it is explicit in Π (i.e., it is given as a fact in the program). Thus, $Claus(Comp'(\Pi))$ will have a clause

$$\neg d_l(\cdots, x, \cdots) \vee x = o \tag{4.41}$$

where $o \in Obj(\Pi)$. From the atom $k(\cdots, f(a_1, \cdots, a_n), \cdots)$ and a series of clauses (4.39), (4.40), $\cdots$, we conclude that $d_l(\cdots, f(a_1, \cdots, a_n), \cdots) \in M'$. Moreover, we conclude that $f(a_1, \cdots, a_n) = o \in M'$ using the clause (4.41). This equality satisfies the instance of $Dca(\Pi)$ for $f(a_1, \cdots, a_n)$ which contradicts with our selection of the term $f(a_1, \cdots, a_n)$. This proves that $f(a_1, \cdots, a_n)$ does not occur in any atom of M' and is insignificant while forming M' (this also applies to our current M). Hence, we can add the equality $f(a_1, \cdots, a_n) = c_x$ to M without affecting extentions of predicates of $Pred(\Pi)$. In order to make M satisfy $Claus(Comp'(\Pi))$, we should substitute c_x by $f(a_1, \cdots, a_n)$ and add the atom $X(f(a_1, \cdots, a_n))$ to M. M satisfies $Claus(Comp'(\Pi))$ since $M \supseteq M'$. Additionally, M satisfies the instance of $Dca(\Pi)$ for $f(a_1, \cdots, a_n)$ with the equality $f(a_1, \cdots, a_n) = c_x$. Since the selection of $f(a_1, \cdots, a_n)$ is arbitrary, we can conclude that M satisfies $Dca(\Pi)$. Also, M coincides with the same answer set A since $M \supseteq M'$. ■

Although one can always add the domain closure axiom to the input theory, it has a negative effect on finite model builders in terms of run time. The finite model builders that we have used in our work have sort inference optimization feature. The domain closure axiom avoids the use of this optimization since the equalities in it prevent the provers infer multiple sorts. The runtime results for blocks world problem in Section 6.4 demonstrate the advantage of expunging the domain closure axiom.

CHAPTER 5

USAGE OF COMPLETOR

Completor[1] is a program that takes a normal logic program as input and outputs a first-order theory which can be used by model generation theorem provers to compute answer sets of the given program. Performance is not the main issue in this beta version of Completor, so there are many possibilities for its optimization and improvement.

As the implementation language, Python was selected for its dynamic nature and high-level data structures, which lead to rapid development.

Completor takes a logic program in a subset of Lparse's language as input. For detailed information on the language of Lparse format, the reader may consult to the manual of Lparse.[2] Extended rules like choice rules with cardinality constraints and weight rules are not supported. Also, arithmetic and comparison operators are not supported yet. Equality is supported in infix notation. Some of the mostly used syntactic features like numeric constant variables, ranges and multiple argument lists are supported. For example, `num(c)` is an atom declaration with a numeric constant variable `c`. While calling Completor one can initialize it to a number value using the `-c` command line option. `num(1..3)` is an atom with a range declaration. It is a shorthand for declaring three atoms; `num(1)`, `num(2)` and `num(3)`. Similarly, `col(red;blue)` is an atom with multiple argument list. It is a shorthand for two atoms; `col(red)` and `col(blue)`.

Completor implements all the transformations defined in Section 4.3 (i.e., *Comp*, *Comp′*, *Claus*, *ElimSkol*, *Una* and *Dca*). The user can specify whether the output is a non-clausal completed theory or a clausified one and whether the Skolem functions in the clausal theory

[1] Completor, http://www.ceng.metu.edu.tr/~orkunt/completor/, visited on June 2009
[2] Lparse 1.0 Users Manual, http://www.tcs.hut.fi/Software/smodels/lparse.ps, visited on June 2009

are eliminated or not. In other words, Completor can complete a logic program, clausify the completed theory and eliminate Skolem functions. It can also infer the sorts from a logic program and generate unique name assumption axioms. Completor generates the output theory in TPTP FOF[3] or TPTP CNF[3] format if the theory is in non-clausal or clausal form respectively.

The following is a list of command line options of Completor:

- `-c <const>=<value>`: The numeric constant variables in a given logic program can be assigned to a value using this option at the command line. This option is useful for generating different instances of a problem. For instance, the number of steps in a planning problem can be represented with a numeric constant variable in the program and the user can set it to various values at the command line.

- `-una None | Naive`: The default is `Naive`. Completor adds the unique name assumption axioms for the set of objects found in a given logic program as described in Definition 4.3.2, by default. Using the `None` option, one can disable adding these axioms.

- `-rn`: If this switch is set, Completor changes all numerical objects in a given logic program by adding a '`n_`' prefix to them. For instance, the constant `2` becomes `n_2` when this switch is set. The reason for changing numbers is that Darwin and FM-Darwin give a syntax error when they are used as numbers in the output format.

- `-clausify None | Defskolem | Defnoskolem`: The default is `None`. By default, Completor does not clausify the input. For the `Defskolem` option, it clausifies the input logic program and there may be Skolem functions introduced. For the `Defnoskolem` option, it again clausifies the input, but Skolem functions are eliminated. For a logic program P, the `None`, `Defskolem` and `Defnoskolem` options are used for constructing theories $Comp(P)$, $Claus(Comp'(P))$ and $ElimSkol(Claus(Comp'(P)))$ respectively. Completor outputs in TPTP FOF format for the `None` option, while it outputs in TPTP CNF format for `Defskolem` and `Defnoskolem` options.

- `-unsorted`: By default, Completor applies the sort detection algorithm for the input logic program in order to generate a theory suitable for the sort optimization feature of finite model builders. However, using this switch disables sort detection and all the

[3] TPTP Syntax, http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html, visited on June 2009

object of the program will be of one sort. Without this switch, when multiple sorts are detected, unique name assumption and totality axioms will be affected as described in Section 4.5.

- `-dca`: Adds the domain closure axiom to the output theory. The domain closure axiom is not added by default, see Corollary 4.3.16 and Section 4.6.

The types of input theories listed in Section 4.4 can be generated using Completor. Consider the logic program P in Example 4.15. It can be written in Lparse format as shown in Figure 5.1.

```
q(2).

r(a,1).
r(b,2).

p(X) :- r(X,Y), not q(Y).
```

Figure 5.1: The logic program P in Lparse format

Let the file containing this program be `example.lp`. The non-clausal input can be generated by the command:

```
$ completor.py example.lp
```

Figure 5.2 shows this non-clausal theory in TPTP FOF format generated by the above command. This input corresponds to (4.16) with the addition of unique name assumption axioms.

The theory in Figure 5.2 is suitable for Paradox, but Darwin and FM-Darwin will not accept it because of the numeric constants used. To form a non-clausal input suitable for Darwin and FM-Darwin, one can use the `-rn` option as in the following command:

```
$ completor.py -rn example.lp
```

The clausal input with Skolem functions which corresponds to (4.23) with the addition of unique name assumption axioms can be generated by the following command:

```
fof(1,axiom,
  ! [U_0]: ( q(U_0) <=> (
        ((U_0=2))) )).
fof(2,axiom,
  ! [U_0,U_1]: ( r(U_0,U_1) <=> (
        ((U_0=a & U_1=1)) |
        ((U_0=b & U_1=2))) )).
fof(3,axiom,
  ! [U_0]: ( p(U_0) <=> (
        (? [Y]: (r(U_0,Y) & ~q(Y)))) )).

% unique name axioms
fof(4,axiom,
  ( $false <= (
        ((1=2))) )).
fof(5,axiom,
  ( $false <= (
        ((a=b))) )).
```

Figure 5.2: The non-clausal theory for program P in TPTP FOF format

```
$ completor.py -clausify Defskolem example.lp
```

Figure 5.3 shows the generated clausal theory in TPTP CNF format. Note that sko_p1 is the introduced Skolem function.

We can eliminate Skolem functions in the clausal theory by typing the following command:

```
$ completor.py -clausify Defnoskolem example.lp
```

It will generate a theory in clausal form where all the Skolem functions are eliminated. The output corresponds to (4.29) with the addition of unique name assumption axioms. This theory in TPTP CNF format is shown in Figure 5.4. The binary predicate skopred_p1 is introduced while eliminating the Skolem function sko_p1.

Notice that Figures 5.2, 5.3 and 5.4 all have two unique name assumption axioms; $1 \neq 2$ and $a \neq b$. Axioms like $1 \neq a$ are eliminated. Completor analyzes the logic program encoding for detecting possible sorts (see Section 4.5). For the program in Figure 5.1, Completor deduces that objects a and b constitute one sort and 1 and 2 constitute another one. This is the reason for eliminating the unique name assumption axioms involving two objects from different sorts.

```
cnf(1,axiom,( q(U_0) | U_0!=2 )).
cnf(2,axiom,( ~q(U_0) | U_0=2 )).
cnf(3,axiom,( r(U_0,U_1) | ~r_r1(U_0,U_1) )).
cnf(4,axiom,( r_r1(U_0,U_1) | U_0!=a | U_1!=1 )).
cnf(5,axiom,( ~r_r1(U_0,U_1) | U_0=a )).
cnf(6,axiom,( ~r_r1(U_0,U_1) | U_1=1 )).
cnf(7,axiom,( r(U_0,U_1) | ~r_r2(U_0,U_1) )).
cnf(8,axiom,( r_r2(U_0,U_1) | U_0!=b | U_1!=2 )).
cnf(9,axiom,( ~r_r2(U_0,U_1) | U_0=b )).
cnf(10,axiom,( ~r_r2(U_0,U_1) | U_1=2 )).
cnf(11,axiom,( ~r(U_0,U_1) | r_r1(U_0,U_1) | r_r2(U_0,U_1) )).
cnf(12,axiom,( p(U_0) | ~r(U_0,Y) | q(Y) )).
cnf(13,axiom,( ~p(U_0) | r(U_0,sko_p1(U_0)) )).
cnf(14,axiom,( ~p(U_0) | ~q(sko_p1(U_0)) )).

% unique name axioms
cnf(15,axiom,( 1!=2 )).
cnf(16,axiom,( a!=b )).
```

Figure 5.3: The clausal theory with Skolem functions for program *P* in TPTP CNF format

```
cnf(1,axiom,( q(U_0) | U_0!=2 )).
cnf(2,axiom,( ~q(U_0) | U_0=2 )).
cnf(3,axiom,( r(U_0,U_1) | ~r_r1(U_0,U_1) )).
cnf(4,axiom,( r_r1(U_0,U_1) | U_0!=a | U_1!=1 )).
cnf(5,axiom,( ~r_r1(U_0,U_1) | U_0=a )).
cnf(6,axiom,( ~r_r1(U_0,U_1) | U_1=1 )).
cnf(7,axiom,( r(U_0,U_1) | ~r_r2(U_0,U_1) )).
cnf(8,axiom,( r_r2(U_0,U_1) | U_0!=b | U_1!=2 )).
cnf(9,axiom,( ~r_r2(U_0,U_1) | U_0=b )).
cnf(10,axiom,( ~r_r2(U_0,U_1) | U_1=2 )).
cnf(11,axiom,( ~r(U_0,U_1) | r_r1(U_0,U_1) | r_r2(U_0,U_1) )).
cnf(12,axiom,( p(U_0) | ~r(U_0,Y) | q(Y) )).
cnf(13,axiom,( ~p(U_0) | r(U_0,Y) | ~skopred_p1(U_0,Y) )).
cnf(14,axiom,( ~p(U_0) | ~q(Y) | ~skopred_p1(U_0,Y) )).
cnf(15,axiom,( skopred_p1(U_0,1) | skopred_p1(U_0,2) )).
cnf(16,axiom,( ~skopred_p1(U_0,Y) | ~skopred_p1(U_0,Y1) | Y=Y1 )).

% unique name axioms
cnf(17,axiom,( 1!=2 )).
cnf(18,axiom,( a!=b )).
```

Figure 5.4: The clausal theory without Skolem functions for program *P* in TPTP CNF format

Also, totality axioms respect sorts of the eliminated Skolem functions (see the 15th clause in Figure 5.4 composed of objects only 1 and 2). In this way finite model builders can take advantage of sort optimization for the generated output. Detection of sorts can be disabled by typing the command:

```
$ completor.py -unsorted -clausify Defnoskolem example.lp
```

This command will output the theory shown in Figure 5.4 except that the totality axiom for skopred_p1 will be

```
cnf(15,axiom,( skopred_p1(U_0,a) | skopred_p1(U_0,b) |
                skopred_p1(U_0,1) | skopred_p1(U_0,2) )).
```

and the unique name assumption axioms will be

```
cnf(17,axiom,( a!=b )).
cnf(18,axiom,( a!=1 )).
cnf(19,axiom,( a!=2 )).
cnf(20,axiom,( b!=1 )).
cnf(21,axiom,( b!=2 )).
cnf(22,axiom,( 1!=2 )).
```

CHAPTER 6

EXPERIMENTS

We carried out a set of experiments on the Ramsey number problem, the pigeon-hole problem, the quasigroup existence problem, and the blocks world planning problem. The experiments aimed to compute one answer set for each program. We did not compute all the answer sets of a program, since the provers we used cannot output more than one model. All the tests in this section except the Ramsey number problem have been performed on a 1.8Ghz Core2Duo machine with 2GB of RAM running Linux kernel. The tests for Ramsey number problem have been performed on a 2.0Ghz Core2Duo machine with 2GB of RAM running Linux kernel.

In [46], we manually formed the input theories for the experiments. In this work, Completor automates this process.

The logic program representations of the benchmark problem instances are first grounded by Lparse and Dlv's grounder. Then, we run answer set solvers on the grounded program in order to compute one answer set. Dlv is an answer set solver for disjunctive logic programs, and there may be disjunctive logic programs for which Dlv exhibits better grounding and search times. However, in the experiments, we also used the same normal logic programs for Dlv.[1] In addition, we run Completor on logic programs with variables to form first-order theories. Completor takes a logic program in a subset of Lparse format as input. It performs all the transformations defined in Section 4.3. It generates the output theory in TPTP format. Next, we use model generation theorem provers in order to find a model of the theory.

[1] Some of the rules of the programs are simplified since Dlv does not restrict all variables in a rule to appear in a domain predicate.

6.1 The Ramsey Number Problem

We performed experiments on the Ramsey number problem of graph theory. Ramsey's theorem states that for any two positive integers m and n, there exists a least positive integer r such that for any undirected complete graph with r vertices, whose edges are colored as red or blue, one can select either a complete subgraph on m vertices which is entirely blue or a complete subgraph on n vertices which is entirely red.[2] Thus, $R(m,n) = r$. We have concentrated on the $R(3,6)$ and $R(4,5)$ problem instances.

Following is the logic program from Asparagus site[3] that is used to encode the Ramsey number problem for the $R(4,5)$ case.

```
arc(X,Y) :- node(X), node(Y), X < Y.                                    (6.1)
```

```
blue(X,Y) :- arc(X,Y), not red(X,Y).
red(X,Y) :- arc(X,Y), not blue(X,Y).

:- blue(W,X), blue(W,Y), blue(X,Y), blue(W,Z), blue(Y,Z),
 blue(X,Z), blue(W,T), blue(X,T), blue(Y,T), blue(Z,T),                 (6.2)
 node(X), node(Y), node(Z), node(W), node(T).

:- red(W,X),red(W,Y),red(X,Y),red(W,Z),red(Y,Z),red(X,Z),
 node(X), node(Y), node(W), node(Z).
```

The rule (6.1) generates an undirected completed graph whose vertices are encoded by the *node* predicate. If there are k *node* atoms and the resulting program has an answer set, it means that k is not the Ramsey number $R(4,5)$ (i.e., $R(4,5) > k$). One can try an instance with $k+1$ nodes.

The last 2 constraint rules in (6.2) have many variables that range over the vertices, which cause them to have many propositional instances. For instance, the first constraint rule in (6.2) has 5 variables and has n^5 propositional instances where n is the number of vertices in the problem. The grounding step of ASP can generate a huge propositional program which makes the job of answer set solvers complicated. Since the model generation theorem provers accept first-order theories, they may show better performance.

[2] Ramsey's theorem, http://en.wikipedia.org/wiki/Ramsey's_theorem/, visited on June 2009

[3] Asparagus, http://asparagus.cs.uni-potsdam.de/, visited on June 2009 This site has benchmark problems and instances for ASP.

For the $R(3,6)$ problem, we tested instances with 12, 13, 14, 15, 16 and 17 vertices. For the $R(4,5)$ case, we tested instances with 19, 20, 21, 22, 23 and 24 vertices.[4] All instances are satisfiable (i.e., have an answer set).

Note that the rule (6.1) uses a comparison operator. The model generation theorem provers that are used do not have a feature of handling arithmetic and comparison operators. For the Ramsey number problem we should explicitly define the comparison operator $<$. The input logic program used by Completor is the same as the above program except (6.1) is replaced by the following rules.[5]

```
arc(X,Y) :- node(X), node(Y), l(X,Y).
```
(6.3)

```
l(X,Y) :- adj(X,Y).
l(X,Y) :- adj(X,Z), l(Z,Y), node(Y).
```
(6.4)

The predicate l in (6.3) is defined for the comparison operator $<$ in (6.1). The intended meaning of an atom $l(i, j)$ is that $i < j$. The adj facts are used for representing the adjacency relation among the numbers that are used to encode nodes. For instance, if there are 3 nodes (i.e., there are 3 facts $node(1)$, $node(2)$, and $node(3)$), the adjacency relation is represented by the facts $adj(1,2)$ and $adj(2,3)$. The rules (6.4) define the less-than relation l by the transitive closure of adj.

It may not be trivial to see that the program used by Completor is tight since the second rule of (6.4) is recursive. The Proposition 6.1.1 states this result formally. Thus, the models found by the model generation theorem provers correspond to the program's answer sets.

Proposition 6.1.1 *Let P be a logic program composed of rules (6.2), (6.3) and (6.4). P also has facts $\{node(i). \mid 1 \leqslant i \leqslant N\}$ and $\{adj(i,j). \mid 1 \leqslant i < N-1,\ 1 < j \leqslant N \text{ and } j = i+1\}$, where N is the number of vertices. The logic program P is tight.*

Proof. We should find a function λ which satisfies the tightness condition. Let $\lambda(node(\_)) = 0$, $\lambda(adj(\_,\_)) = 0$, $\lambda(arc(\_,\_)) = 2N$, $\lambda(blue(\_,\_)) = 2N+1$ and $\lambda(red(\_,\_)) = 2N+1$.[6] For

[4] $R(3,6) = 18$ and $R(4,5) = 25$.

[5] Note that we used the original program, which have the rule (6.1), in the tests with answer set solvers.

[6] $\lambda(adj(\_,\_)) = 0$ denotes that λ assigns every ground adj atom to 0.

ground instances of l, let $\lambda(l(i,j)) = 2N-(i+j)$. Any ground instance of the first rule of (6.4) is of the following form.

```
l(i,i+1) :- adj(i,i+1).
```
(6.5)

Since the minimum value of $\lambda(l(\_,\_))$ is 1 (for the atom $l(N-1,N)$, $\lambda(l(N-1,N)) = 1$), the tightness condition is satisfied for (6.5) ($\lambda(l(i,i+1)) > \lambda(adj(i,i+1))$ where $\lambda(adj(i,i+1)) = 0$). Any ground instance of the second rule of (6.4) is of the following form.

```
l(i,j) :- adj(i,i+1), l(i+1,j), node(j).
```
(6.6)

Using the previous reasoning, we get that $\lambda(l(i,j))$ is less than both $\lambda(adj(i,i+1))$ and $\lambda(node(j))$. $\lambda(l(i,j)) > \lambda(l(i+1,j))$ since $2N-(i+j) > 2N-(i+j+1)$. Thus, (6.6) satisfies the tightness condition. In order to see that (6.3) satisfies the tightness condition, it is sufficient to observe that the maximum value of $\lambda(l(\_,\_))$ is $2N-2$ since $\lambda(l(1,1)) = 2N-2$. It is easy to see that all the rules in (6.2) satisfy the tightness condition. Hence, it follows that the logic program P is tight. ■

Table 6.1: The run times of Smodels, Cmodels, Clasp and Dlv on the Ramsey number problem. The entries are total CPU times in seconds; '–' means that the solver could not produce an answer within a time limit of 600 seconds. The 'Grounding' column gives the CPU times of grounding the problem instances by Lparse. There are two values in Dlv entries: the first one gives the grounding time, while the second gives the search time.

Problem	Grounding	Smodels	Cmodels	Clasp			Dlv
R(3,6) 12	54.5	44.2	33.1	18.6	0.2	/	0.1
R(3,6) 13	88.9	70.9	54.6	32.7	0.1	/	0.1
R(3,6) 14	139.9	118.4	–	54.4	0.2	/	0.2
R(3,6) 15	218.1	–	–	91.7	0.3	/	0.4
R(3,6) 16	–				0.5	/	0.7
R(3,6) 17	–				0.6	/	1.1
R(4,5) 19	33.5	29.8	22.3	15.1	0.5	/	0.9
R(4,5) 20	43.1	38.7	28.3	20.0	0.7	/	1.2
R(4,5) 21	55.4	111.9	36.3	26.1	0.9	/	1.6
R(4,5) 22	70.2	62.5	46.5	34.0	1.2	/	2.1
R(4,5) 23	86.3	–	–	44.0	1.4	/	6.5
R(4,5) 24	106.6	–	–	–	1.9	/	–

Table 6.1 shows the run times of Smodels (version 2.32), Cmodels (version 3.70), Clasp (version 1.01), and Dlv (version 2006-07-14) on Ramsey number problem. The time values are the total CPU times in seconds corresponding to the sum of user and system time values

given by the shell's *time* command.[7] The run times of this problem show that grounding is a bottleneck for the answer set solvers Smodels, Cmodels, and Clasp. Lparse (version 1.0.17) spends too much memory for all of the instances. Lparse begins trashing (i.e., the operating system starts swaping in and out memory for the user process and the user process cannot do anything useful) for the $R(3,6)$ 15 instance. Furthermore, it cannot ground the instances $R(3,6)$ 16 and $R(3,6)$ 17 within the time limit. Answer set solvers that use Lparse could not be tested for these two instances. Additionally, the run times of grounding are significant compared to search times. For Cmodels and Clasp, grounding takes more time than searching for an answer set. All these results show that Lparse generates a huge amount of propositional instances of the program. This hardens the search process of solvers. The larger instances of the problems $R(3,6)$ and $R(4,5)$ could not be solved within the time limit.

Dlv displayed very good performance for this problem compared to the other solvers. It is basically related to the grounding done by Dlv. Grounding times of Dlv are very fast compared to Lparse. The efficient grounding is also reflected in good search performance of Dlv.

Table 6.2 supports the claim that Lparse generates a huge amount of propositional instances of the program more clearly. The number of rules in the ground program generated by Lparse represents how large the propositional instance of the program is. For instance, numbers of rules generated for the $R(3,6)$ 15 and $R(4,5)$ 24 instances are approximately 11 and 8 millions respectively. When we compare the number of rules generated by Lparse and Dlv, we can definitely claim that Dlv performs better grounding than Lparse for this problem. Since Paradox uses a SAT solver to build a finite model, we include the number of variables and clauses generated at the last step which Paradox finds a model or proves that the input is unsatisfiable. Recall that finite model building has an incremental nature where finite domain sizes are increased gradually. Unfortunately, Paradox 3.0 has no option for outputting this kind of information. The numbers in Table 6.2 are from tests done with an old version of Paradox (version 1.3) which outputs related information. Although it may not be possible to directly compare Paradox 3.0 and Paradox 1.3 in this sense, we put this information just to give an idea of how large the generated propositional theory is. Additionally, comparison between Paradox and Lparse/Dlv by only size information may not expose clear and reliable

[7] Since the evaluation machine has a multi-core CPU, we used the sum of user and system time values instead of real time value. The real time may be less than the user time since the programs of shell pipes in the test scripts and other programs like eprover called within a prover may run on different cores in parallel.

results because of the differences in the inner workings of the theorem prover and answer sets solvers. Other than that, complexities of ground rules and propositional clauses may differ. For instance, Completor + Paradox displays a better performance than Dlv for this problem, although the propositional theory generated by Paradox is larger than the ground program generated by Dlv.

Table 6.2: Sizes of the ground programs generated by Lparse or Dlv and sizes of the propositional theories generated by Paradox 1.3 for the Ramsey number problem instances. The '#atoms' and '#rules' columns give the number of atoms and rules respectively in the ground programs generated by Lparse or Dlv. Similarly, the '#vars' and '#clauses' columns are for the number of variables and clauses respectively in the propositional theories generated by Paradox 1.3. The numbers related to Paradox 1.3 are the ones corresponding to finite domains for which Paradox finds a model or proves that the input is unsatisfiable.

	Lparse		Dlv		Paradox 1.3	
Problem	#atoms	#rules	#atoms	#rules	#vars	#clauses
R(3,6) 12	367	2987922	211	1354	4666	7689
R(3,6) 13	430	4829253	248	2249	5809	28200
R(3,6) 14	498	7532567	288	3654	7124	56978
R(3,6) 15	571	11394330	331	5790	8623	96513
R(3,6) 16	–	–	377	8944	11148	150929
R(3,6) 17	–	–	426	13481	13746	223489
R(4,5) 19	913	2606952	533	16036	18594	155532
R(4,5) 20	1011	3360590	591	20939	21393	199598
R(4,5) 21	1114	4279233	652	26985	24458	252198
R(4,5) 22	1222	5388603	716	34364	27317	273974
R(4,5) 23	1335	6716966	783	43286	30376	367423
R(4,5) 24	1453	8295252	853	53982	34217	519340

Neither Darwin (version 1.4.4) nor FM-Darwin (version 1.4.4) could solve any instance of the Ramsey number problem within the time limit. [8] states that Darwin is weak for problems with equality. Axiomatic equality reasoning can be a bottleneck for Darwin. FM-Darwin transforms the problem to a satisfiability problem over function-free clause logic without equality [7]. It uses Darwin as an engine to build a finite model. Since this transformation eliminates equality, the inefficient equality reasoning alone cannot actually explain the weak results of FM-Darwin. The transformation used in FM-Darwin introduces new predicate symbols and adds new clauses for the totality constraints needed over the finite domain. These increase the size of the theory and the search space, which, in turn, makes the job of Darwin more complicated.

Table 6.3: The run times of Paradox on the Ramsey number problem. The entries are total CPU times in seconds; '–' means that the prover did not halt within a time limit of 600 seconds. The 'Completor' column gives the run times of Completor for completing the logic program, clausifying the completed theory and eliminating Skolem functions (i.e., generating the 'no Skolem' type of input). The 'non-clausal' column is for non-clausal first-order theory formed by completing the logic program. Both the 'clausal' and 'no Skolem' columns are for the completed and clausified theories, but the Skolem functions introduced during clausification are eliminated in the latter.

			Paradox	
Problem	Completor	non-clausal	clausal	no Skolem
R(3,6) 12	0.2	0.1	0.1	0.2
R(3,6) 13	0.2	0.1	0.2	0.2
R(3,6) 14	0.2	0.2	0.2	0.3
R(3,6) 15	0.2	0.2	0.3	0.3
R(3,6) 16	0.2	0.3	0.3	0.4
R(3,6) 17	0.2	0.4	0.4	0.5
R(4,5) 19	0.2	0.3	0.4	0.5
R(4,5) 20	0.2	0.5	0.6	0.7
R(4,5) 21	0.2	0.7	0.8	1.0
R(4,5) 22	0.2	1.0	1.1	1.3
R(4,5) 23	0.2	3.2	2.7	3.1
R(4,5) 24	0.3	553.9	–	–

Table 6.3 shows the run time results of Paradox (version 3.0) on Ramsey number problem. Completor completes the logic program with variables and outputs three types of theories. Figure 6.1 depicts paths of the applied transformations for generating these three types of input. The results of the corresponding non-clausal first-order theories are shown in the *non-clausal* column of Table 6.3. This type of input corresponds to $Comp(P) \cup Una(P)$ where P is the logic program. The *clausal* column represents the results of the inputs that are formed by clausification of the completed theory. They correspond to $Claus(Comp'(P)) \cup Una(P)$. They may contain Skolem functions introduced during clausification. Note that, as a consequence of Theorem 4.6.3, there is no need to add the domain closure axiom to the non-clausal theory and the clausal one with Skolem functions for this problem, since the logic program is domain-restricted. This also holds for each of the benchmark problems in this section. Completor can also eliminate these Skolem functions and produce another input. The *no Skolem* column shows results of this type of input, which corresponds to $ElimSkol(Claus(Comp'(P))) \cup Una(P)$.

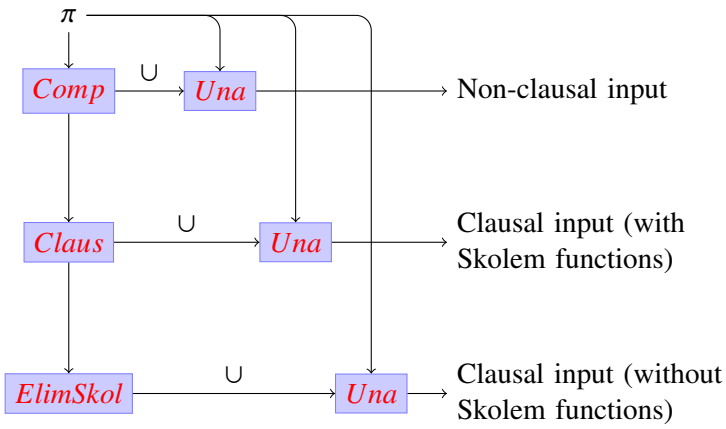

Figure 6.1: Paths of transformation applications for generating various types of input theories

Paradox solved every instance of the problem instantly.[8] It displayed nearly the same performance for three types of input (i.e., inputs in non-clausal form, clausal form with Skolem functions, and clausal form without Skolem functions). The results compared to the ones in Table 6.1 display the clear advantage of Completor and Paradox instead of answer set solvers. For instance, in order to find an answer set of the instance $R(3,6)$ 15 using Clasp, it takes 218.1 + 91.7 = 309.8 seconds of processing time (grounding + search times). We can find an answer set of the same instance in only 0.2 + 0.3 = 0.5 seconds using Completor and Paradox. Table 6.4 shows these total processing time results when Completor and Paradox are used together compared to the total processing time results of answer set solvers. Although the performance difference between Paradox and Dlv is not as marginal as the one between Paradox and the other solvers that use Lparse as a grounder, Paradox has better results than those of Dlv. Hence, Completor + Paradox is the best system for this experiment.

Note that Paradox also performs grounding before calling the SAT solver, but this is done incrementally. It calls the SAT solver for every finite domain starting with the one of size 1 and increases incrementally (see Section 3.2). It is interesting to compare Cmodels and Paradox since Cmodels is a SAT-based answer set solver. The order of grounding and completion steps is interchanged in Paradox and Cmodels. Figure 6.2 compares Cmodels and Paradox from this perspective. The input of Paradox is the completed first-order theory formed by Completor, whereas the input is the grounded logic program in Cmodels. Paradox performs the grounding, while Cmodels performs the completion. The incremental nature of finite

[8] Except the instance $R(4,5)$ 24, but Paradox is the only system that found an answer set for this instance.

Table 6.4: The total processing times of answer set solvers and Paradox on the Ramsey number problem. For answer set solvers, the total processing time is the sum of grounding and search times. For Paradox, it is the sum of Completor time and search time.

Problem	Smodels	Cmodels	Clasp	Dlv	Paradox no Skolem
R(3,6) 12	98.7	87.6	73.1	0.3	0.4
R(3,6) 13	159.8	143.5	121.6	0.2	0.4
R(3,6) 14	258.3	–	194.3	0.4	0.5
R(3,6) 15	–	–	309.8	0.7	0.5
R(3,6) 16				1.2	0.6
R(3,6) 17				1.7	0.7
R(4,5) 19	63.3	55.8	48.6	1.4	0.7
R(4,5) 20	81.1	71.4	63.1	1.9	0.9
R(4,5) 21	167.3	91.7	81.5	2.5	1.2
R(4,5) 22	132.7	116.7	104.2	3.3	1.5
R(4,5) 23	–	–	130.3	7.9	3.3
R(4,5) 24	–	–	–	–	–

model building and the other optimizations of Paradox seem to be efficient for this problem.

Additionally, we tested Cmodels using Minisat[9] as its SAT solver since Paradox uses Minisat. The run time performance of Cmodels with Minisat was worse than or approximately the same as that of Cmodels with zChaff. Thus, the efficiency of Paradox compared to Cmodels should not be related with just Minisat but with our method of answer set computation and the incremental nature of finite model building.

6.2 The Pigeon-Hole Problem

The logic program of this problem is attributed to Ilkka Niemelä [41]. It encodes the basic pigeon-hole principle: placing P pigeons into H holes so that there is at most one pigeon in a hole. Following is the logic program in Lparse format.

```
pigeon(1..p).
hole(1..h).

pos(P,H) :- pigeon(P), hole(H), not negpos(P,H).
```

[9] Cmodels can use Minisat with a command line option. The default SAT solver is zChaff.

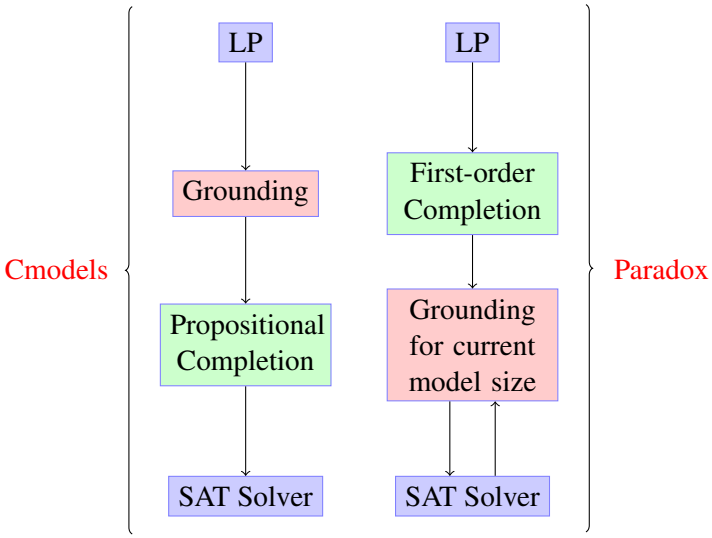

Figure 6.2: Comparison of Cmodels and Paradox in terms of the order of completion and grounding

```
negpos(P,H) :- pigeon(P), hole(H), not pos(P,H).

hashole(P) :- pigeon(P), hole(H), pos(P,H).
:- pigeon(P), not hashole(P).

:- pigeon(P), hole(H), hole(H1), pos(P,H), pos(P,H1), H != H1.
:- pigeon(P),  pigeon(P1), hole(H),  pos(P,H), pos(P1,H), P != P1.
```

The logic program is a tight one since there is a function λ which satisfies the tightness condition by assigning each instance of the *pigeon* and *hole* predicates to 0, each instance of the *pos* and *negpos* predicates to 1, and each instance of the *hashole* predicate to 2.

We tested 9 instances: 5 consistent and 4 inconsistent ones. The instances with 50, 70, 90, 110 and 130 pigeons and holes are satisfiable. Those with 8, 9, 10 and 11 pigeons but 7, 8, 9 and 10 holes, respectively, have no answer sets. In the tables, the instances are named by the numbers of pigeons and holes; for example, 10-9 refers to the instance with 10 pigeons and 9 holes.

In general, unsatisfiable instances of the pigeon-hole problem are tested as a benchmark. However, in our work we also tested satisfiable instances. In the satisfiable case, this prob-

lem may seem to be trivial. But we can increase the number of pigeons and holes to form large instances. The propositional instances of the logic programs for problems with a large number of pigeons and holes will also be large. This has effects on grounding and answer set computation. One reason for testing satisfiable instances is to examine these effects. Another is to see whether completing the programs without grounding and running model generation theorem provers will improve the performance of computing answer sets or not.

Table 6.5: The run times of Smodels, Cmodels, Clasp and Dlv on the pigeon-hole problem. The 'S/U' column gives whether the instance is satisfiable or not. The structure of the table is analog to Table 6.1.

Prb.	S/U	Grnd.	Smodels	Cmodels	Clasp			Dlv
50-50	s	1.1	53.2	2.0	1.1	2.7	/	34.4
70-70	s	3.1	393.0	6.5	3.3	8.8	/	214.4
90-90	s	6.6	–	13.3	7.6	22.1	/	–
110-110	s	12.0	–	24.4	14.6	50.0	/	–
130-130	s	19.8	–	42.3	25.0	108.7	/	–
8-7	u	0.01	0.4	0.1	0.1	0.01	/	1.6
9-8	u	0.01	3.7	0.6	0.6	0.01	/	15.3
10-9	u	0.01	36.3	3.3	4.2	0.02	/	161.6
11-10	u	0.02	394.8	7.6	34.2	0.02	/	–

Table 6.5 shows the run times of Smodels, Cmodels, Clasp, and Dlv on the pigeon-hole problem instances. The grounding times of the satisfiable instances show that the logic program with variables has a large propositional instance. The run times of grounding are not insignificant anymore, since they are comparable to the search times of the solvers for the satisfiable instances. Dlv's grounder did not scale up well. This may be due to the normal logic program encoding of the problem, since Dlv shows the best performance for disjunctive programs. Smodels and Dlv were not able to solve the instances with 90, 110 and 130 pigeons within the time limit. In general, Smodels and Dlv showed weak performance for the pigeon-hole problem. Clasp is the fastest answer set solver for the satisfiable instances and Cmodels is the fastest for the unsatisfiable ones.

Table 6.6 shows the sizes of ground programs and propositional theories generated by Lparse, Dlv and Paradox 1.3. Since Lparse and Dlv has generated ground instances with same number of atoms and rules, the results are shown in the same column. The large number of rules of ground programs for satisfiable instances explain why the run times of grounding in Table 6.5

are significant compared to the search run times for these instances.

Table 6.6: Sizes of the ground programs generated by Lparse or Dlv and sizes of the propositional theories generated by Paradox 1.3 for the pigeon-hole problem instances. The structure of the table is analog to Table 6.2.

	Lparse/Dlv		Paradox 1.3	
Problem	#atoms	#rules	#vars	#clauses
50-50	5151	252650	14137	300997
70-70	10011	691110	26797	812333
90-90	16471	1466370	43457	1700457
110-110	24531	2674430	64894	3073996
130-130	34191	4411290	90590	5052788
8-7	136	919	406	1272
9-8	171	1322	502	2108
10-9	210	1829	608	2519
11-10	253	2452	724	3734

Table 6.7: The run times of FM-Darwin, Paradox and Darwin on the pigeon-hole problem. The 'S/U' column gives whether the instance is satisfiable or not. The structure of the table is analog to Table 6.3.

			FM-Darwin	Paradox			Darwin
Prb.	S/U	Completor	no Skolem	non-clausal	clausal	no Skolem	no Skolem
50-50	s	0.5	–	1.1	1.0	0.8	–
70-70	s	0.8	–	2.5	2.2	2.0	–
90-90	s	1.2	–	4.6	4.2	3.6	–
110-110	s	1.8	–	7.7	7.0	6.4	–
130-130	s	2.5	–	11.7	11.0	9.9	–
8-7	u	0.1	11.5	–	–	0.1	5.3
9-8	u	0.1	71.7	–	–	0.6	27.1
10-9	u	0.1	–	–	–	3.8	139.7
11-10	u	0.1	–	–	–	65.2	–

Table 6.7 shows the run time results of the model generation theorem provers on the pigeon-hole problem instances. The clausal theory formed after eliminating Skolem functions is in the EPR problem class. Paradox and FM-Darwin are refutationally complete for this class. When the given program instance is unsatisfiable, we can only consider the theories of type *no Skolem* for Paradox or FM-Darwin. For instance, Paradox found the unsatisfiability of

the instances 8-7, 9-8, 10-9 and 11-10 as depicted in the *no Skolem* column. Similarly, FM-Darwin found the unsatisfiability of the instances 8-7 and 9-8. Note that in the columns *non-clausal* and *clausal*, where the input is not in the EPR class, Paradox was not able to halt for the unsatisfiable instances.

Table 6.7 shows that Paradox exhibits the best performance for satisfiable instances of the pigeon-hole problem compared to the answer set solvers and other model generation theorem provers. The run times for the input of type *no Skolem* are slightly better those for the inputs of types *non-clausal* and *clausal*. The run times of Completor for generating input theories of type *no Skolem* are also shown in Table 6.7. It should be recalled that, the grounding times in Table 6.5 are no longer insignificant compared to the search times. In addition to good search time performance, if we consider the grounding times, the performance difference of Paradox compared to answer set solvers increases. For instance, a problem with 130 pigeons can be solved in 44.8 seconds (19.8 + 25.0; grounding and search times) using Clasp. The same instance can be solved in 12.4 seconds (2.5 + 9.9; Completor and search times) using Paradox. Table 6.8 shows these total processing time results. For the unsatisfiable instances, Paradox exhibited a similar performance to Cmodels or Clasp and outperformed Smodels and Dlv.

Table 6.8: The total processing times of answer set solvers and Paradox on the pigeon-hole problem. For answer set solvers, the total processing time is the sum of grounding and search times. For Paradox, it is the sum of Completor time and search time.

Prb.	S/U	Smodels	Cmodels	Clasp	Dlv	Paradox no Skolem
50-50	s	54.3	3.1	2.2	37.1	1.3
70-70	s	396.1	9.6	6.4	223.2	2.8
90-90	s	–	19.9	14.2	–	4.8
110-110	s	–	36.4	26.6	–	8.2
130-130	s	–	62.1	44.8	–	12.4
8-7	u	0.4	0.1	0.1	1.6	0.2
9-8	u	3.7	0.6	0.6	15.3	0.7
10-9	u	36.3	3.3	4.2	161.6	3.9
11-10	u	394.8	7.6	34.2	–	65.3

Both FM-Darwin and Darwin showed weak performances for the pigeon-hole problem. The results of FM-Darwin for theories of type *non-clausal* and *clausal* are not depicted in Table

6.7, since FM-Darwin was not able to solve any of these theories within the time limit.

6.3 The Quasigroup Existence Problem

We also performed experiments on the quasigroup existence (QG) problem[10] in algebra [40]. We selected the QG2, QG5 and QG7 variations. QG problems impose constraints on a relation whose multiplication table is an order n Latin square. The QG2 problem's constraint on the relation $\circ$ is that if $a \circ b = c \circ d \wedge b \circ e = a \wedge d \circ e = c$, then $a = c \wedge b = d$ for all a, b, c, d, and e in an order n Latin square defined for $\circ$. The QG5 problem's constraint is $((a \circ b) \circ a) \circ a = b$. The constraint for the QG7 problem is $(b \circ a) \circ b = a \circ (b \circ a)$. Additionally, in all of the variations the relation $\circ$ should also satisfy the constraint $a \circ a = a$. Figure 6.3 depicts a solution of the QG5 problem of order 5.

$\circ$	1	2	3	4	5
1	1	4	5	2	3
2	5	2	4	3	1
3	2	1	3	5	4
4	3	5	1	4	2
5	4	3	2	1	5

Figure 6.3: A solution of the QG5 problem of order 5

Following is a tight logic program in Lparse format that is used to encode the QG2 problem. The tightness condition is satisfied by the existence of a function λ which assigns each instance of the *range* predicate to 0, each instance of the *val* and *nval* predicates to 1, and each instance of the *hasval* predicate to 2.

```
range(1..n).

val(X,Y,Z) :- not nval(X,Y,Z), range(X;Y;Z).
nval(X,Y,Z) :- not val(X,Y,Z), range(X;Y;Z).
:- val(X,Y,Z), val(X,Y,Z1), Z != Z1, range(X;Y;Z;Z1).
```

[10] The definition and variations are described in the problem prob003 from CSPLIB (Prob003: Quasigroup Existence, http://www-old.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob003/index.html, visited on June 2009).

```
hasval(X,Y) :- val(X,Y,Z), range(X;Y;Z).
:- not hasval(X,Y), range(X;Y).

:- val(X,Y,Z), val(X,Y1,Z), Y != Y1, range(X;Y;Z;Y1).
:- val(X,Y,Z), val(X1,Y,Z), X != X1, range(X;Y;Z;X1).

:- val(X,X,X1), X != X1, range(X;X1).

% QG2 constraint
:- val(A,B,X), val(C,D,X), val(B,E,A), val(D,E,C), A!=C,
   range(A;B;C;D;E;X).
:- val(A,B,X), val(C,D,X), val(B,E,A), val(D,E,C), B!=D,
   range(A;B;C;D;E;X).
```

We tested the instances of orders 5 to 11 for each problem. While the satisfiable instances (i.e., instances for which a corresponding quasigroup exists) are marked as 's' mark in instance names in the tables, the unsatisfiable instances are marked as 'u'.

Table 6.9 shows the run times of Smodels, Cmodels, Clasp, and Dlv on the QG instances. None of the systems solved the instances of orders 10 and 11 within the time limit (except the instance of order 11 for QG5 problem). Smodels and Dlv could not scale up well for the QG problems as the order increases. Clasp had better results than Cmodels.

Table 6.10 shows the sizes of ground programs and propositional theories generated by Lparse, Dlv, and Paradox 1.3. The sizes for QG7 instances are not shown since they are the same as the sizes for QG5 instances. The sizes of ground programs generated by Lparse and Dlv indicate that the propositional instances of the QG problems (except QG2) are not as large as those of previous benchmark problems. However, the rules of the logic program may still have many and complex instances since they have many variables that range over the numbers from 1 to n, where n is the order of the problem instance [48]. For instance, the last 2 constraint rules in the logic program encoding of the QG2 problem have 6 variables each. The number of propositional instances of each constraint rule is close to n^6. The QG2 part of the Table 6.10 shows that the numbers of rules of ground programs become very large as the order increases. Furthermore, the grounding times of Lparse and Dlv for the QG2 problem,

Table 6.9: The run times of Smodels, Cmodels, Clasp and Dlv on the QG problems. The 'S/U' column gives whether the instance is satisfiable or not. The structure of the table is analog to Table 6.1.

Prb.	S/U	Grnd.	Smodels	Cmodels	Clasp		Dlv
QG2 5	s	0.2	0.2	0.1	0.1	0.4 /	0.6
6	u	0.6	5.5	0.6	0.3	1.0 /	19.8
7	s	1.6	3.6	1.6	0.9	2.8 /	28.7
8	s	3.7	–	–	28.3	6.0 /	–
9	s	7.5	–	–	–	13.1 /	–
10	u	14.4	–	–	–	24.5 /	–
11	s	26.4	–	–	–	44.5 /	–
QG5 5	s	0.04	0.02	0.02	0.02	0.1 /	0.1
6	u	0.1	0.2	0.1	0.1	0.1 /	0.3
7	s	0.1	0.2	0.1	0.1	0.3 /	0.4
8	s	0.2	4.1	0.3	0.2	0.6 /	0.9
9	u	0.4	–	–	223.2	1.2 /	–
10	u	0.7	–	–	–	2.2 /	–
11	s	1.2	–	17.5	2.2	4.0 /	–
QG7 5	s	0.03	0.02	0.02	0.02	0.04 /	0.1
6	u	0.1	0.4	0.1	0.1	0.1 /	0.5
7	u	0.1	24.4	1.1	0.5	0.3 /	50.4
8	u	0.2	–	35.3	58.5	0.6 /	–
9	s	0.4	372.3	1.1	0.6	1.2 /	–
10	u	0.7	–	–	–	2.2 /	–
11	u	1.2	–	–	–	3.9 /	–

which are displayed in Table 6.9, are not insignificant. For example, it took 26.4 seconds for Lparse to ground the QG2 instance of order 11. Paradox 1.3 has generated propositional theories whose sizes are smaller than the sizes of the ground programs generated by Lparse or Dlv.

Table 6.11 shows the run time results of FM-Darwin, Paradox, and Darwin on the QG instances. Paradox has scaled up better than Smodels, Dlv, and Cmodels. Additionally, the run time results of Paradox are similar to those of Clasp. Table 6.12 shows the total processing time results. Note that, one should only consider the *no Skolem* column when testing finite model builders with unsatisfiable input. Similarly, the *no Skolem* column should be considered when testing Darwin with satisfiable input. Only the results for clausal input without Skolem functions (i.e., the results in *no Skolem* columns) are displayed for FM-Darwin and Darwin, since they are same as or better than the ones for the other two types of input.

The results depicted in Table 6.11 show that both FM-Darwin and Darwin have weak performances for this problem. The comments that were stated for the previous experimental problems also apply to this one.

Table 6.10: Sizes of the ground programs generated by Lparse or Dlv and sizes of the propositional theories generated by Paradox 1.3 for the QG problem instances. The sizes for the QG7 problem are not shown since they are equal to the sizes for the QG5 problem in both cases (i.e., Lparse/Dlv and Paradox 1.3). The structure of the table is analog to Table 6.2.

	Lparse/Dlv		Paradox 1.3	
Prb.	#atoms	#rules	#vars	#clauses
QG2 5	281	26925	1073	5368
6	475	81720	1792	11247
7	743	208985	2777	20946
8	1097	471168	4070	35833
9	1549	964629	5713	57512
10	2111	1830200	7748	84843
11	2795	3265185	10217	125736
QG5 5	281	4425	698	3182
6	475	10440	1144	6522
7	743	21707	1748	11291
8	1097	41088	2534	19663
9	1549	72333	3526	31830
10	2111	120200	4748	48814
11	2795	190575	6224	71731

6.4 The Blocks World Problem

The blocks world program is a planning problem where blocks are moved to form a desired configuration. There can be stacks of blocks on a table. One can only move the block which is on top of a stack. The following logic program in Lparse format is attributed to Ilkka Niemelä [41]. It has been shown in [4] that the models of its completion correspond to its answer sets (i.e., it is tight on its models of completion).

```
% goal predicate which holds when the goal conditions have be reached
goal :- time(T), goal(T).
:- not goal.
```

Table 6.11: The run times of FM-Darwin, Paradox and Darwin on the QG problems. The 'S/U' column gives whether the instance is satisfiable or not. The structure of the table is analog to Table 6.3.

			FM-Darwin	Paradox			Darwin
Prb.	S/U	Completor	no Skolem	non-clausal	clausal	no Skolem	no Skolem
QG2 5	s	0.2	0.2	0.04	0.04	0.04	1.2
6	u	0.2	–	–	–	0.1	310.3
7	s	0.2	485.4	0.1	0.4	0.1	–
8	s	0.2	–	3.2	5.2	3.0	–
9	s	0.2	–	–	–	–	–
10	u	0.2	–	–	–	–	–
11	s	0.2	–	–	–	–	–
QG5 5	s	0.2	0.2	0.03	0.04	0.03	0.2
6	u	0.2	–	–	–	0.04	9.1
7	s	0.2	77.7	0.1	0.1	0.1	103.6
8	s	0.2	–	0.1	0.1	0.1	–
9	u	0.2	–	–	–	288.4	–
10	u	0.2	–	–	–	–	–
11	s	0.2	–	44.2	43.9	17.6	–
QG7 5	s	0.2	0.2	0.03	0.03	0.03	0.3
6	u	0.2	–	–	–	0.1	19.8
7	u	0.2	–	–	–	0.2	337.9
8	u	0.2	–	–	–	22.5	–
9	s	0.2	–	0.2	1.2	0.4	–
10	u	0.2	–	–	–	–	–
11	u	0.2	–	–	–	–	–

```
goal(T2) :- nextstate(T2,T1), goal(T1).

% operator preconditions
moveop(X,Y,T):- time(T), block(X), object(Y), X != Y,
        on_something(X,T), available(Y,T), not covered(X,T),
        not covered(Y,T), not blocked_move(X,Y,T).

% operator effects
on(X,Y,T2) :- block(X), object(Y), nextstate(T2,T1), moveop(X,Y,T1).
on_something(X,T) :- block(X), object(Z), time(T), on(X,Z,T).

available(table,T) :- time(T).
```

Table 6.12: The total processing times of answer set solvers and Paradox on the QG problems. For answer set solvers, the total processing time is the sum of grounding and search times. For Paradox, it is the sum of Completor time and search time.

Prb.	S/U	Smodels	Cmodels	Clasp	Dlv	Paradox no Skolem
QG2 5	s	0.4	0.3	0.3	1.0	0.2
6	u	6.1	1.2	0.9	20.8	0.3
7	s	5.2	3.2	2.5	31.5	0.3
8	s	–	–	32.0	–	3.2
9	s	–	–	–	–	–
10	u	–	–	–	–	–
11	s	–	–	–	–	–
QG5 5	s	0.1	0.1	0.1	0.2	0.2
6	u	0.3	0.2	0.2	0.4	0.2
7	s	0.3	0.2	0.2	0.7	0.3
8	s	4.3	0.5	0.4	1.5	0.3
9	u	–	–	223.6	–	288.6
10	u	–	–	–	–	–
11	s	–	18.7	3.4	–	17.8
QG7 5	s	0.1	0.1	0.1	0.1	0.2
6	u	0.5	0.2	0.2	0.6	0.3
7	u	24.5	1.2	0.6	50.7	0.4
8	u	–	35.5	58.7	–	22.7
9	s	372.7	1.5	1.0	–	0.6
10	u	–	–	–	–	–
11	u	–	–	–	–	–

```
available(X,T) :- block(X), time(T), on_something(X,T).
covered(X,T) :- block(Z), block(X), time(T), on(Z,X,T).

% frame axioms
on(X,Y,T2) :- nextstate(T2,T1), block(X), object(Y),
        on(X,Y,T1), not moving(X,T1).
moving(X,T) :- time(T), block(X), object(Y), moveop(X,Y,T).

% Stop applying operators when the goal has been reached
blocked_move(X,Y,T):- block(X), object(Y), time(T), goal(T).

% moveop(X,Y,T) is blocked if its not chosed
```

```
blocked_move(X,Y,T) :- time(T),block(X),object(Y), not moveop(X,Y,T).

% An object can be moved by one move operation at a time
blocked_move(X,Y,T) :- block(X), object(Y), object(Z),
        time(T), moveop(X,Z,T), Y != Z.

% An object cannot be moved on top of an object that is moved
blocked_move(X,Y,T) :- block(X), object(Y), time(T), moving(Y,T).

% Two objects cannot be moved on top of the same object
% by one move operation at a time
blocked_move(X,Y,T) :- block(X;Y;Z), time(T), moveop(Z,Y,T), X != Z.

% pruning rules
:- block(X), time(T), moveop(X,table,T), on(X,table,T).
:- nextstate(T2,T1), block(X), object(Y),
        moveop(X,Y,T1), moveop(X,table,T2).

% facts
time(0..steps).
nextstate(Y,X) :- time(X), time(Y), Y = X + 1.
object(table).
object(X) :- block(X).
```

The experiments take into account 4 blocks world problem instances: an instance encoding of the Susman anomaly case (bw1) with 3 blocks, instances that correspond to the problems PLA026+1 (bw2) and PLA025+1 (bw3) from the TPTP library with 5 and 9 blocks, respectively, and the large.c instance (bw4) from [41] with 15 blocks. They have plans of 3, 3, 4 and 8 steps, respectively.

Table 6.13 shows the run times of Smodels, Cmodels, Clasp, and Dlv[11] on blocks world problems. All the solvers solved the problems instantly, even with the addition of grounding

[11] The inputs for Dlv are from http://www.dbai.tuwien.ac.at/proj/dlv/examples/tests-bw.tar.gz, visited on June 2009 .

Table 6.13: The run times of Smodels, Cmodels, Clasp and Dlv on blocks world problems. The structure of the table is analog to Table 6.1.

Problem	Grounding	Smodels	Cmodels	Clasp		Dlv
bw1	0.01	0.01	0.01	0.01	0.01 /	0.01
bw2	0.02	0.01	0.01	0.01	0.02 /	0.03
bw3	0.1	0.1	0.1	0.04	0.1 /	0.2
bw4	0.4	1.4	0.5	0.3	1.5 /	2.1

times of Lparse or Dlv.

Table 6.14: Sizes of the ground programs generated by Lparse or Dlv and sizes of the propositional theories generated by Paradox 1.3 for the blocks world problem instances. The structure of the table is analog to Table 6.2.

	Lparse		Dlv		Paradox 1.3	
Prb.	#atoms	#rules	#atoms	#rules	#vars	#clauses
bw1	224	693	267	588	3428	8689
bw2	476	2149	537	1856	13191	41456
bw3	1580	11577	1682	10369	78310	258132
bw4	7106	81729	7320	75563	525833	1818599

This problem has a small propositional instance. Table 6.14 supports this claim. We have selected this benchmark problem in order to compare the model generation theorem provers with answer set solvers in terms of run time performance when grounding generates small propositional instances. We expect that using answer set solvers will be advantageous since our proposed method of computing answer sets using model generation systems is best suited to problems that have huge amount of or complex propositional instances.

Table 6.15 shows the run times of FM-Darwin, Paradox, and Darwin on blocks world problems. The run times of Darwin are provided only for the *no Skolem* column since Darwin may not halt for the other two input types. The run time results show that none of the model generation theorem provers tested scale up well for the blocks world problem compared to the answer set solvers. This is in accordance with our expectation which is stated above in the analysis related to the sizes of the propositional instances.

Table 6.15: The run times of FM-Darwin, Paradox and Darwin on blocks world problems. The structure of the table is analog to Table 6.3.

		FM-Darwin			Paradox			Darwin
Prb.	Completor	non-clausal	clausal	no Skolem	non-clausal	clausal	no Skolem	no Skolem
bw1	0.5	2.8	0.3	0.3	0.1	0.1	0.2	0.5
bw2	0.8	34.2	12.8	21.5	0.4	0.3	0.4	10.0
bw3	1.2	–	–	–	8.6	6.8	3.1	–
bw4	1.8	–	–	–	–	438.7	47.4	–

6.4.1 The Effect of Disabling Sort Optimization

The encoding of the blocks world problem is suitable for sort inference optimization of finite model builders. Time values and block names (including table) constitute important concepts of the problem. These concepts can be considered as different sorts. This will help in searching for a model since the size of its finite domain is less than the size of the unsorted one.[12] In order to take advantage of sort optimization, Completor generates unique name assumption axioms for each sort (i.e., the axioms $\{a \neq table,\ a \neq b,\ \cdots\} \cup \{0 \neq 1,\ 0 \neq 2,\ \cdots\}$ where inequalities such as $a \neq 0$ are eliminated. Note that $a, b, c, \cdots$ are block names and $0, 1, 2, \cdots$ are time points in the planning problem). Otherwise, finite model builders will infer only one sort. Additionally, when eliminating Skolem functions from a clausal theory, totality axioms should respect the sort information. We also tested the blocks world problem with unsorted input theories (using the `-unsorted` option of Completor) to see the advantage of sort inference optimization of finite model builders. The results in Table 6.16 compared to ones in Table 6.15 depict the big drawback of using a theory for which Paradox or FM-Darwin cannot infer multiple sorts. For instance, it took 248 seconds for Paradox to find a model of the bw4 instance for the unsorted case while it took 47.4 seconds for the multi-sorted case.

6.4.2 The Effect of Adding the Domain Closure Axiom

Table 6.17 shows the run times of finite model builders when we add the domain closure axiom to inputs (using the `-dca` option of Completor). The equalities in the domain closure

[12] If there are 5 blocks in a problem instance and we are searching for a 9-step plan, then 9 domain elements are enough to find a model of the problem. Some of the domain elements can be shared for the interpretations of the block and step constants.

Table 6.16: The run times of FM-Darwin and Paradox on blocks world problems where input theories are unsorted. The structure of the table is analog to Table 6.3.

	FM-Darwin			Paradox		
Prb.	non-clausal	clausal	no Skolem	non-clausal	clausal	no Skolem
bw1	6.3	1.2	1.3	0.5	0.6	0.8
bw2	106.8	24.1	32.3	5.1	4.3	1.9
bw3	–	–	–	170.9	69.3	12.2
bw4	–	–	–	–	–	248.0

axiom may cause the finite model builders infer only one sort, since equalities involving in the domain closure axiom covers all the objects in the program. This may be disadvantageous in terms of run time. Fortunately, there is no need for the addition of the domain closure axiom as stated in Theorem 4.6.3 and Corollary 4.3.16. Actually, the results in Table 6.17 compared to ones in Table 6.15 show this disadvantage and the practical importance of Theorem 4.6.3 and Corollary 4.3.16. For instance, it took 10.1 seconds for Paradox to find a model of the bw3 instance when the domain closure axiom had been added while it took 6.8 seconds for the case without the domain closure axiom. For the bw4 instance it is even worse; Paradox could not find a model of it within the time limit when the domain closure axiom had been added. In terms of run time, FM-Darwin was affected much worse than Paradox did by the addition of the axiom.

Table 6.17: The run times of FM-Darwin and Paradox on blocks world problems where input theories include the domain closure axiom. The structure of the table is analog to Table 6.3.

	FM-Darwin			Paradox		
Prb.	non-clausal	clausal	no Skolem	non-clausal	clausal	no Skolem
bw1	9.4	3.3	4.3	0.2	0.2	0.2
bw2	262.8	434.1	219.2	0.4	0.4	0.4
bw3	–	–	–	16.6	10.1	3.3
bw4	–	–	–	–	–	71.5

CHAPTER 7

CONCLUDING REMARKS

This thesis proposes a method of computing answer sets of normal logic programs using model generation theorem provers as computational engines. This method can be seen as lifting SAT-based ASP to the first-order level for tight programs. The completion step in the lifted case produces a first-order theory. Replacing the SAT solver with a system which can process first-order input and produce a model when it is satisfiable is fundamental to this study. The model generation theorem provers are suitable for this role. One important outcome of this lifting is that grounding can either be eliminated or performed more intelligently in the model generation prover.

Model generation theorem provers are gaining increasing popularity in the automated reasoning community. We used Darwin, FM-Darwin, and Paradox in our experiments. Related with ASP, Darwin was used in [17] for testing strong equivalence of logic programs with variables under answer set semantics.

In the scope of this thesis, a program named Completor was implemented to generate input theories for model generation theorem provers. Completor implements the necessary transformations to generate a first-order theory from normal logic programs. Some of these transformations are needed because of completeness properties of provers; while some of them are needed because of semantic distinctions between first-order logic and ASP. Briefly, Completor can complete a normal logic program with variables, clausify the completed theory, and eliminate Skolem functions. It can also infer sorts from a logic program to take advantage of sort optimization of finite model builders.

The empirical analysis showed that the use of Completor and model generation systems to-

gether, can be an effective way of computing answer sets. The run time results of Paradox showed that using Completor and Paradox is favorable compared to answer set solvers in the Ramsey number, pigeon-hole, and quasigroup existence problems. Lparse generated large propositional programs for the Ramsey number problem and the satisfiable instances of the pigeon-hole problem which made the job of the answer set solvers difficult. It was clearly observed that, for the Ramsey number problem, grounding could be a bottleneck since some instances of the problem could not be grounded. Besides, the grounding times were not insignificant compared to the search times for such programs. The run times of Paradox for these problems displayed the advantage of lifting SAT-based ASP to the first-order level more clearly. Both Darwin and FM-Darwin exhibited weak performance on our benchmark problems. Axiomatic equality reasoning, which is used by Darwin, is inefficient. The equality literals generated by completion and the unique name assumption axioms increase the importance of equality reasoning.

Although Darwin has shown weak performance in our experiments, the ME calculus, on which Darwin is based, is interesting and can be beneficial for future answer set solvers working on the first-order level. Note that the search procedures of successful answer set solvers, like Smodels and Dlv, are based on the DPLL procedure at the propositional level and the ME calculus is a lifting of the DPLL procedure to the first-order level.

Paradox and FM-Darwin are refutationally complete for theories in the EPR class. The clausal theories formed in this work by eliminating Skolem functions (i.e., by the *ElimSkol* transformation) are in the EPR class. The input theories in the EPR class should be used for solving inconsistent problems; otherwise, the finite model builder may not halt. The relation between the EPR formulas and logic programs under the answer set semantics has been examined in [32] for the application area of textual inference. That work defines a translation from any EPR formula to a logic program such that the formed logic program has an answer set iff the EPR formula is satisfiable. In [42], EPR class is used as a language to represent problems. Then, solvers that natively handle EPR formulas (like Darwin or Paradox) are used to solve problems.

A definition of the stable models of an arbitrary first-order formula is given in [21]. This work also extends the tightness condition to first-order formulas. Using that definition, it can be decided whether a logic program is tight or not without referring to the original tightness

condition. The tightness condition based on Fages' theorem refers to grounding [19], but the new one does not. Thus, one can check whether a program is tight or not without grounding it.

One limitation of the provers that we used is that they can only produce one model for a given input theory. The search algorithm of Darwin can be modified to find all models in such a way that, after a model is found, it can backtrack to the last application of the Split rule and continue with the right side of the branching point formed by that Split rule. For the case of finite model builders (Paradox and FM-Darwin), symmetric models that correspond to the same answer set should be eliminated while searching for all models.

Another limitation is that provers used cannot reason with arithmetic and comparison operators. This may prevent problems from being represented with a logic program in an easy and natural way. Facts about these operators for a finite set of integers can be explicitly added to the program (for instance, adding facts $plus(1,1,2)$, $plus(1,2,3)$, $plus(2,1,3), \cdots$ for defining the + operator for numbers $\{1,2,3\}$). However, this can be disadvantageous in terms of run time performances since large number of facts are needed. In Section 6.1 we have overcome this limitation with a succinct definition of less-than operator.

We have concentrated on Mace-style finite model building. One future line of research will be examining the Sem-style finite model builders like Sem[1] and Finder.[2] A Sem-style finite model builder searches for a model by working on the input directly. Unlike Mace-style finite model building, there is no translation of the problem into a satisfiability problem [13].

[1] Finite model generation, http://www.cs.uiowa.edu/~hzhang/sem.html, visited on June 2009
[2] Finder page, http://users.rsise.anu.edu.au/~jks/finder.html, visited on June 2009

REFERENCES

[1] C. Anger, K. Konczak, T. Linke, and T. Schaub. A glimpse of answer set programming. *Künstliche Intelligenz*, 19:12–17, 2005.

[2] K. R. Apt. Logic programming. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 493–574, 1990.

[3] K. R. Apt, A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[4] Y. Babovich, E. Erdem, and V. Lifschitz. Fages' theorem and answer set programming. *In Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, 2000.

[5] Y. Babovich and V. Lifschitz. Computing answer sets using program completion. *Unpublished draft*, 2003.

[6] M. Balduccini and M. Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3:425–461, 2003.

[7] P. Baumgartner, A. Fuchs, H. de Nivelle, and C. Tinelli. Computing Finite Models by Reduction to Function-Free Clause Logic. *In Proceedings of the 3rd Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability at the 3rd International Joint Conference on Automated Reasoning*, 2006.

[8] P. Baumgartner, A. Fuchs, and C. Tinelli. Implementing the Model Evolution Calculus. *International Journal of Artificial Intelligence Tools*, 15:21–52, 2006.

[9] P. Baumgartner and G. Stenz. Instance based methods. Tutorial at the Second International Joint Conference on Automated Reasoning, 2004.

[10] P. Baumgartner and C. Tinelli. The Model Evolution Calculus. *Fachberichte Informatik 1–2003, Universität Koblenz-Landau, Institut für Informatik*, 2003.

[11] P. Baumgartner and C. Tinelli. The Model Evolution Calculus with Equality. *In Proceedings of the 20th International Conference on Automated Deduction*, 2005.

[12] A. K. Chandra and D. Harel. Horn clause queries and generalization. *Journal of Logic Programming*, 2:1–15, 1985.

[13] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. *In Proceedings of Workshop: Model Computation – Principles, Algorithms, Applications at the 19th International Conference on Automated Deduction*, 2003.

[14] K. Clark. Negation as failure. *In Logic and Data Bases*, pages 293–322, 1978.

[15] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[17] T. Eiter, W. Faber, and P. Traxler. Testing Strong Equivalence of Datalog Programs - Implementation and Examples. *In Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science 3662*, pages 437–441, 2005.

[18] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[19] F. Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[20] C. G. Fermüller and R. Pichler. Model Representation via Contexts and Implicit Generalizations. *In Proceedings of the 20th International Conference on Automated Deduction, Lecture Notes in Computer Science 3632*, pages 409–423, 2005.

[21] P. Ferraris, J. Lee, and V. Lifschitz. A New Perspective on Stable Models. *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 372–379, 2007.

[22] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. *In Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 386–392, 2007.

[23] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):619–649, 1991.

[24] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. *In Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, 1988.

[25] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–386, 1991.

[26] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

[27] T. Hinrichs and M. Genesereth. Herbrand Logic. Technical report, LG-2006-02, 2006.

[28] S. Jacobs and U. Waldmann. Comparing instance generation methods for automated reasoning. *Journal of Automated Reasoning*, 38(1-3):57–78, 2007.

[29] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[30] R. A. Kowalski. Predicate logic as programming language. *In Proceedings of International Federation of Information Processing Conference*, pages 569–574, 1974.

[31] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2006.

[32] Y. Lierler and V. Lifschitz. Logic Programs vs. First-Order Formulas in Textual Inference. *Unpublished draft*.

[33] Y. Lierler and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. *In Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 346–350, 2004.

[34] V. Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pages 69–127, 1996.

[35] V. Lifschitz. Answer set planning. *In Proceedings of the 16th International Conference on Logic Programming*, pages 23–37, 1999.

[36] V. Lifschitz and H. Turner. Splitting a Logic Program. *In Proceedings of the 11th International Conference on Logic Programming*, pages 23–37, 1994.

[37] F. Lin and Y. Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157:115–137, 2004.

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

[39] V. Marek and M. Truszczynski. Stable models and an alternative logic programming paradigm. *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398, 1999.

[40] W. McCune. A Davis-Putnam Program and Its Application to Finite First-Order Model Search: Quasigroup Existence Problems. *Technical Report ANL/MCS-TM-194, Argonne National Laboratory*, 1994.

[41] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

[42] J. A. Navarro Pèrez. *Encoding and Solving Problems in Effectively Propositional Logic*. PhD thesis, The University of Manchester, 2007.

[43] T. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[44] R. Reiter. Equality and domain closure in first-order databases. *Journal of the ACM*, 27(2):235–249, 1980.

[45] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[46] O. Sabuncu and F. N. Alpaslan. Computing Answer Sets Using Model Generation Theorem Provers. *In Proceedings of the 4th Workshop on Answer Set Programming Advances in Theory and Implementation at the 23rd International Conference on Logic Programming*, pages 225–240, 2007.

[47] O. Sabuncu, F. N. Alpaslan, and V. Akman. Using Criticalities as a Heuristic for Answer Set Programming. *In Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Artificial Intelligence 2923*, pages 234–246, 2004. A Workshop version appeared In Proceedings of the 2nd International Workshop on Answer Set Programming: Advances in Theory and Implementation, Messina, Italy.

[48] Y. Shirai and R. Hasegawa. Answer Set Computation Based on a Minimal Model Generation Theorem Prover. *In Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence*, pages 43–52, 2004.

[49] P. Simons, I. Niemelá, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[50] T. Soininen and I. Niemelä. Developing a Declarative Rule Language for Applications in Product Configuration. *In Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319, 1999.

[51] G. Sutcliffe and S. Melville. The practice of clausification in automatic theorem proving. *South African Computer Journal*, 18, 1996.

[52] T. Syrjänen. Omega-restricted programs. *In Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning, Lecture Notes in Computer Science 2173*, pages 267–280, 2001.

[53] T. Tammet. Finite model building: improvements and comparisons. *In Proceedings of Workshop: Model Computation – Principles, Algorithms, Applications at the 19th International Conference on Automated Deduction*, 2003.

APPENDIX A

ESTABLISHING THE TEST ENVIRONMENT

A test environment is necessary for performing the experiments whose results are shown in Chapter 6. This appendix gives brief information about the programs and scripts which are needed and describes how to establish the test environment. This environment is formed on Linux operating system.

Firstly, we have installed the answer set solvers and the grounder Lparse. Recall that the answer set solvers Smodels, Cmodels, and Clasp all use Lparse as the grounder. Dlv has an internal grounder. The following is a brief information about getting and installing the mentioned systems. More detailed information can be found in Section 2.3.

- SMODELS (http://www.tcs.hut.fi/Software/smodels/). It is developed at Helsinki University of Technology. The source package can be downloaded from its website. It is implemented in C++. One can easily compile it by a `make` command on a standard Linux distribution.

- CMODELS (http://www.cs.utexas.edu/users/tag/cmodels.html). It is developed at University of Texas, Austin. It is implemented in C++ and should be compiled by a `make` command on a Linux machine.

- CLASP (http://www.cs.uni-potsdam.de/clasp/). It is developed at University Potsdam. It is implemented in C++. Although there are pre-compiled Linux executables for some versions of Clasp, one may have to compile it on a Linux machine to work with newer versions.

- DLV (http://www.dbai.tuwien.ac.at/proj/dlv/). It is developed at TU Wien. There are pre-compiled executables for Linux on its website. Dlv has an internal grounder. In

order to find grounding run times when using Dlv, we have used the `-instantiate` command line option. For instance, the following command instantiates the input program `example.lp` and writes its propositional instance to `in.grnd`.

```
$ dlv -instantiate example.lp > in.grnd
```

Unlike Lparse, Dlv uses the `-N` option to set a maximum integer value that can be used in a input program. Additionally, we should explicitly state that we are searching for only one answer set by using the `-n=1` switch.

- LPARSE (http://www.tcs.hut.fi/Software/smodels/). Lparse is developed at Helsinki University of Technology. Lparse is a grounder which forms a propositional instance of an input program. Smodels, Cmodels, and Clasp all use Lparse as a grounder. Lparse takes input program files as command line arguments and writes the propositional one to the standard output. One of the mostly used command line option of Lparse is the `-c` switch which assigns values to numeric constant variables in the input logic program. Using this switch, one can easily generate multiple instances of the problem using the same logic program representation. The following command instantiates the program `qg2.lp` representing the quasigroup existence problem (QG2). Lparse instantiates the QG2 problem of order 8 with the help of `-c` switch. Later, the output program `in.grnd` can be input to Smodels, Cmodels or Clasp.

  ```
  $ lparse -c n=8 qg2.lp > in.grnd
  ```

Next, we have installed the model generation theorem provers and Completor. For the experiments, Completor is used to generate input theorems for these theorem provers. Detailed information about the Completor system can be found in Chapter 5.

- DARWIN (http://combination.cs.uiowa.edu/Darwin/). Darwin is a theorem prover for first-order logic. It is implemented in OCAML. In order to compile an executable of Darwin, OCAML compiler, interpreter and libraries should be installed. The website of Darwin has pre-compiled Linux executables for all of its versions. It accepts input theorems in TPTP and TME formats. Since it works with clauses, a non-clausal input theorem should first be converted to clausal logic. If the input is in non-clausal form, Darwin uses eprover as a pre-processing system to clausify the input. The following

command shows the basic use of Darwin. The -pmd switch is used to output the found model in DIG format.

```
$ darwin_linux_v1.4.4 -pmd true example.tptp
```

- FM-DARWIN (http://combination.cs.uiowa.edu/Darwin/). FM-Darwin is a finite model builder for first-order logic. It comes bundled with Darwin. Actually, one can use FM-Darwin by calling Darwin with the -fd command line argument. Thus, one can install FM-Darwin by installing Darwin. The following command shows the basic use of FM-Darwin. The -pmfd switch is used to output the found finite model.

```
$ darwin_linux_v1.4.4 -fd true -pmfd true example.tptp
```

- PARADOX (http://www.cs.chalmers.se/~koen/folkung/). Paradox is a finite model builder for first-order logic. It is mainly implemented in Haskell. It uses Minisat as a SAT solver while searching for finite models. Minisat is included in the source distribution of Paradox and it is implemented in C++. In order to compile an executable of Paradox, Haskell compiler and libraries should be installed in the system. Fortunately, the website offers a pre-compiled Linux executable. Paradox accepts inputs in TPTP format. When the input theorem is not in clausal form, it first clausifies the input. Unlike Darwin, Paradox has an internal clausifier. It outputs the found finite model when the -model switch is set in the command line. The following command shows the basic use of Paradox.

```
$ paradox3 -model example.tptp
```

  Unfortunately, the latest version of Paradox (version 3.0) has no command line option for outputting the size of the instantiated propositional theory corresponding to the finite domain for which Paradox finds a model or proves that the input is unsatisfiable. This kind of information can be used to compare the sizes of propositional logic programs generated by grounders and propositional theories generated by Paradox. An old version of Paradox (version 1.3) outputs this kind of information.

- COMPLETOR (http://www.ceng.metu.edu.tr/~orkunt/completor/). We developed Completor within the scope of this thesis. The detailed usage information can be found in

Chapter 5. Completor is implemented in Python. You need a Python interpreter to install and use Completor. Recall that Completor accepts input logic programs in a subset of Lparse language. The parser for input logic programs is developed with the help of Ply package for Python. Ply[1] is basically an implementation of Lex and Yacc parser tools for Python. Ply is included in the source distribution of Completor.

The output of Completor is a theory in TPTP format. The ouput theory can consist of formulas or clauses. Then, the output theory is given to the model generation theorem provers as input. In order to ease testing we formed shell scripts for each theorem prover. These scripts are responsible for forming a theory from a logic program using Completor and for calling the related theorem prover with right options. Additionally, these scripts use the shell `time` command to measure CPU times spent by Completor and the theorem prover.

The call scripts are basically Bash shell scripts. In many Linux distributions the support for Bash scripts comes by default. The following is an example call script for Paradox named as `complete_and_paradox`. The variables `PARADOX_BIN` and `COMPLETOR` are paths for executables of Paradox and Completor respectively. The call script saves all the intermediate work such as the initial logic program, the formed theory in TPTP format in `tests` directory for later analysis. Basically, this script takes arguments to be used with Completor, name template for saved intermediate files and at least one input logic program file.

```
#!/bin/bash

PARADOX_BIN=/home/orkunt/systems/paradox3
COMPLETOR=/home/orkunt/systems/scripts/completor/completor.py
TESTSDIR="tests"

USAGE="Usage: complete_and_paradox ('completor arguments'|none)
        input_name_template files*"

completor_args=""
if [ "$1" = "none" ]; then
        shift
```

[1] Ply (Python Lex-Yacc), http://www.dabeaz.com/ply/, visited on June 2009

```
else
        completor_args=$1
        shift
fi

input_name_template=$1
shift

if [ "$#" = 0 ]; then
        echo $USAGE
        exit $?
fi

if [ ! -d $TESTSDIR ]; then
        mkdir $TESTSDIR
fi

COMPLETOR_OUT=$TESTSDIR/$input_name_template.tptp
INPUT_FILE=$TESTSDIR/$input_name_template.lp

cat "$@" > $INPUT_FILE
echo "Input:" $INPUT_FILE >&2

completor_command="$COMPLETOR $completor_args  $INPUT_FILE"
echo $completor_command" > $COMPLETOR_OUT" >&2
time $completor_command > $COMPLETOR_OUT

echo $PARADOX_BIN --model $COMPLETOR_OUT >&2
time $PARADOX_BIN --model $COMPLETOR_OUT

exit $?
```

Similarly, we have prepared call scripts for each of the answer set solvers. These call scripts

are responsible for calling Lparse to ground the program (for Dlv, the call script calls Dlv) and for calling the related solver. They also use the shell `time` command to measure CPU times spent. Below is a call script for Smodels.

```
#!/bin/bash

SMODELS_BIN=/home/orkunt/bin/smodels
LPARSE_BIN=/home/orkunt/bin/lparse

if [ -z "$1" ]; then
        echo "usage: callsmodels (-lparsep "parameter") [files]*"
        exit 0
fi

lparse_param=""
# check if there is a lparse parameter
if [ "$1" = "-lparsep" ]; then
        lparse_param=$2
        shift 2
fi

cmd="time cat "$@" | $LPARSE_BIN $lparse_param > in.grnd"
echo "Grounding " $cmd >&2
time cat "$@" | $LPARSE_BIN $lparse_param > in.grnd
cmd="time $SMODELS_BIN < in.grnd"
echo "Testing " $cmd >&2
time $SMODELS_BIN < in.grnd

#time cat "$@" | $LPARSE_BIN $lparse_param | $SMODELS_BIN

exit $?
```

After preparing the call scripts, we prepared scripts for each of the benchmark experiments.

These scripts basically call call scripts with necessary arguments and input logic program files. For instance, the following is an example script for Ramsey number problem which tests Paradox with clausal input where Skolem functions are eliminated. The script uses the `TIMELIMIT` variable to denote the time limit value. The `ulimit` command is used to restrict that the call script is executed for `TIMELIMIT` amount of CPU time at most. The input files for the Ramsey number problem are `ramsey3-6_completor.lp` and `ramsey4-5_completor.lp` for the $R(3,6)$ and $R(4,5)$ problems respectively. Additionally, input files for each instance of the problem are given as input to the call script `complete_and_paradox` (for instance, `adj12`). Note that `-clausify Defnoskolem` option is specified so that Completor generates clausal theories where Skolem functions are eliminated.

```
#!/bin/bash

# Pigeon hole problem with clausal theory
# (Skolem functions are eliminated)

INPUT1=ramsey3-6_completor.lp
INPUT2=ramsey4-5_completor.lp
TIMELIMIT=600
PREFIX=ramsey_px_defnoskol
SCRIPT=/home/orkunt/systems/scripts/testscripts/complete_and_paradox

ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_12" adj12 $INPUT1;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_13" adj13 $INPUT1;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_14" adj14 $INPUT1;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_15" adj15 $INPUT1;
```

```
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_16" adj16 $INPUT1;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_36_17" adj17 $INPUT1;
ulimit -S -t unlimited

ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_19" adj19 $INPUT2;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_20" adj20 $INPUT2;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_21" adj21 $INPUT2;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_22" adj22 $INPUT2;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_23" adj23 $INPUT2;
ulimit -S -t unlimited
ulimit -S -t $TIMELIMIT;
$SCRIPT '-clausify Defnoskolem' $PREFIX"_45_24" adj24 $INPUT2;
ulimit -S -t unlimited
```

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Sabuncu, Orkunt

Nationality: Turkish (TC)

Data and Place of Birth: 12 January 1978, Denizli

Marital Status: Single

Phone: +90 312 2105511

Fax: +90 312 2105544

Email: orkunt@ceng.metu.edu.tr

EDUCATION

Degree	Institution	Year of Graduation
MS	METU Computer Engineering	2002
BS	METU Computer Engineering	1999
High School	Denizli Anatolian High School	1995

WORK EXPERIENCE

Year	Place	Enrollment
2008-Present	ORBİM Software Corporation	Technical Manager
2006-2007	METU ISC-Excellency Center	Research assistant
2005-2006	AGMLAB	Software engineer
2002-2005	METU Computer Engineering Dept.	Research assistant
1999-2001	KOÇSİSTEM	Software developer specialist
1997-1998	TÜBİTAK-BİLTEN	Part-time programmer

PUBLICATIONS

O. Sabuncu and F. N. Alpaslan, "Computing Answer Sets Using Model Generation Theorem Provers," *Information Sciences,* (on review).

D. Tunaoğlu, Ö. Alan, O. Sabuncu, S. Akpınar, N. Çiçekli and F. N. Alpaslan, "Event Extraction from Turkish Football Web-casting Texts Using Hand-crafted Templates," *In Proceedings of the 3rd IEEE International Conference on Semantic Computing ICSC,* 2009 (forthcoming).

Ö. Alan, S. Akpınar, O. Sabuncu, N. Çiçekli and F. N. Alpaslan, "Ontological Video Annotation and Querying System for Football Games," *In Proceedings of the 23rd International Symposium on Computer and Information Sciences ISCIS,* 2008.

O. Sabuncu and F. N. Alpaslan, "Computing Answer Sets Using Model Generation Theorem Provers," *In Proceedings of the 4th Workshop on Answer Set Programming Advances in Theory and Implementation at the 23rd International Conference on Logic Programming ICLP,* 2007.

O. Sabuncu, F. N. Alpaslan and V. Akman, "Using Criticalities as a Heuristic for Answer Set Programming," *In Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning LPNMR,* 2004.

O. Sabuncu, F. N. Alpaslan and V. Akman, "Using Criticalities as a Heuristic for Answer Set Programming," *In Proceedings of Answer Set Programming Workshop ASP,* 2003.