GROUNDING FOR MODEL EXPANSION IN K-GUARDED FORMULAS WITH INDUCTIVE DEFINITIONS

by

Murray Patterson

Bachelor of Computer Science with Honours, Acadia University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

APPROVAL

Name:	Murray Patterson
Degree:	Master of Science
Title of thesis:	Grounding for Model Expansion in k-Guarded Formulas with Inductive Definitions

Examining Committee: Mr. Bradley Bart, Lecturer, Computing Science
Simon Fraser University
Chair

Dr. Arvind Gupta, Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Eugenia Ternovska, Assistant Professor, Computing Science
Simon Fraser University
Co-Senior Supervisor

Dr. Andrei Bulatov, Assistant Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: August 3, 2006

Abstract

Mitchell and Ternovska [49, 50] propose a constraint programming framework for search problems that is based on classical logic extended with inductive definitions. They formulate a search problem as the problem of model expansion (MX). In this framework, the problem is encoded in a logic, an instance of the problem is represented by a finite structure, and a solver generates solutions to the problem. This approach relies on propositionalisation of high-level specifications, and on the efficiency of modern SAT solvers. Here, we propose an efficient algorithm which combines grounding with partial evaluation. Since the MX framework is based on classical logic, we are able to take advantage of known results for the so-called guarded fragments and their generalizations. In the case of k-guarded formulas with inductive definitions under a natural restriction, the algorithm performs much better than naive grounding by relying on connections between k-guarded formulas and tree decompositions.

Keywords grounding; model expansion; guarded fragments; descriptive complexity; inductive definitions

Acknowledgments

First, I would like to thank Eugenia Ternovska, my senior supervisor for her dedication and teaching me more about what research is like at a professional level. Second, I would like to thank Arvind Gupta, my senior supervisor for keeping me on track and keeping me from getting lost in details. I also thank Andrei Bulatov for his helpful comments on the final document. Finally, I especially thank Yongmei Liu for her insight and knowledge on this subject. This thesis would not be what it is today without her.

Contents

List of Figures

Chapter 1

Introduction

The well-known theorem of Fagin states that existential second-order logic ($\exists$SO) captures the complexity class **NP** [17]. This implies that any problem in **NP** can be expressed in $\exists$SO, that is, we are able to encode any computational task with this inherent time bound. The implication of this theorem and other results in descriptive complexity theory is logics can be viewed as programming languages for their corresponding complexity classes (see, e.g. [15]). While Fagin's theorem, and other results in descriptive complexity are very meaningful in this way, much remains to be done to make this idea practical, *i.e.*, to develop an approach to declarative constraint programming that is efficient, conceptually clean, and has good modeling capabilities. This thesis makes a step towards this goal.

The most successful approaches in the past to declarative constraint programming are propositional satisfiability (SAT), constraint satisfaction problems (CSP), and answer set programming (ASP).

SAT is the oldest and most developed of the three approaches and today is both the subject of research and used by industry for solving specific problems. A SAT problem is simply represented by a boolean formula, where solving it involves determining whether there is a satisfying assignment to the formula. Since a satisfying assignment in this context is just a set of truth values for all of the boolean variables that sets the formula to *true*, the semantics of SAT is very simple. It is this semantic simplicity that is the key driver in the success of SAT for modeling problems. A drawback of SAT is that it lacks the modeling capabilities of logics with quantifiers or recursion. An even bigger drawback of SAT is the lack of separation between problem instance and problem description. For example, given the problem of determining whether or not a graph is 3-colorable, and an instance (a graph) we can construct a propositional formula that has a satisfying assignment for every 3-coloring of the graph (if any exist). However, this representation is specific to this problem/instance pair. It cannot be reused, or even trivially modified in most cases, to solve 3-colorability for any other graph. This lack of separation leads to conceptual and practical difficulties.

CSP provides somewhat better modeling capabilities than SAT. One reason for this is that variables need not be boolean; instead, variables may take values from any specified finite domain. It also provides some techniques used to solve problems, such as nogood learning and backjumping methods, which are now central to modern SAT solvers [51]. Solvers for CSP, however, have found practical success primarily as components in constraint logic programming (CLP) tools. These provide rich problem solving environments, often used to produce domain-specific solvers for **NP**-hard problems. However, they are general-purpose programming languages, and thus not purely declarative.

Another approach to declarative constraint programming that emerged from logic programming is Answer Set Programming (ASP) (see [4]). The semantics of ASP are based on the stable model semantics of Gelfond and Lifschitz [19], and, as such, it is much more complex than SAT or CSP. ASP was proposed as a programming paradigm in [52, 47]. The main advantage of ASP is that it has a better modeling language than SAT or CSP, e.g., it allows for some implicit use of quantifiers and has a built-in recursion mechanism. ASP has a more clear separation of instance and problem description than SAT, but not a complete separation, as instance and problem description are separate only on the level of methodology. The main disadvantage of ASP is that non-determinism can only be imitated formally via recursion through negation. Because of this, some features of classical logic cannot be modeled in ASP in a natural way. For example, in classical logic, a boolean variable p is free to be true or false. However to represent this in ASP, the programmer must introduce a new atom p', and add rules $p \leftarrow \mathit{not}\ p'$ and $p' \leftarrow \mathit{not}\ p$ to represent this same feature.

While results in descriptive complexity suggest that problems of a given complexity can be expressed in a certain logic, there exists no purely declarative constraint programming framework for hard problems that is close in syntax or semantics, to the corresponding logics that can express these problems. This comes as no surprise because it is not obvious how to build a general-purpose tool for modeling problems in a highly declarative fashion.

However, as a step in this direction, Mitchell and Ternovska [49, 50] proposed a declarative constraint programming framework, which is based on classical logic extended with non-monotone inductive definitions. They cast a search problem as the classical problem of *model expansion* (MX), which is the problem of expanding a given structure with new relations so that it satisfies a given formula. The long-term goal is to develop tools for solving hard combinatorial search problems for different complexity classes, especially those in **NP**. In this framework, the problem description is encoded in a logic and a problem instance is represented by a finite structure, where some of the predicates in the formula are not specified in the structure. A solver then generates solutions (if any exist) to a given model expansion problem by finding structures that *expand* the instance structure (*i.e.*, provides interpretations for the initially unspecified, or expansion, predicates) in such a way that these expansion structures satisfies the formula. To our knowledge, this framework is the first declarative constraint programming framework to be based on finite model theory and descriptive complexity.

The MX framework combines the many strengths of SAT, CSP and ASP, as well as addresses their limitations. Its features include: a high-level modeling language that supports quantification and recursion, and a clear separation of the instance (a finite structure) from the problem description (a formula). Most importantly, the framework achieves the goal of developing a tool for modeling problems with a highly declarative and natural language based on classical logic extended with inductive definitions. This makes it possible to exploit the many existing results from finite model theory and descriptive complexity.

Note that in the case of first-order (FO) logic, the MX problem is exactly the same as model-checking (MC) for $\exists$SO. In fact, finding interpretations for unspecified predicates in an MX problem for FO is the same as witnessing $\exists$ for the corresponding $\exists$SO problem. As such, the MX problem for FO captures **NP** as an easy corollary of Fagin's theorem. Furthermore, since inductively defined properties are hard to represent in FO logic, extending FO logic with inductive definitions (IDs) to get FO(ID) logic allows for more convenience in modeling problems. While the MX framework is a general framework that deals with MX problems for many logics, in this thesis, we obtain results for the FO and FO(ID) cases only, and as such, we concern ourselves only with these logics. Throughout the thesis, we may use MX when we mean MX for FO or FO(ID) logic.

The advances in the efficiency of propositional solvers over the past few years has led researchers to use it as a component in high level solvers, e.g. [8, 9, 14]. It has also led researchers in the area of ASP to build solvers that *propositionalize* (or *ground*) the program and solve these using a propositional solver, for example [43, 45]. Since ASP is based on logic programming, and thus resembles the propositional satisfiability problem even less than FO logic (or even FO(ID) logic) in both syntax and semantics, this is strong evidence that the MX framework can be made practical by this approach of grounding followed by the use of a propositional solver. For one thing, with ASP, grounding must be performed over the Herbrand universe, however, in the MX framework, since the instance to the problem is a finite structure, it is clear that every term must be an element of the domain of this structure, *i.e.*, the grounding will always be finite. In particular, for MX, we want to ground the FO or FO(ID) formula over the structure in a way that this grounding has a satisfying assignment for every expansion of the instance structure that satisfies the FO(ID) formula (if any exist). In the case of MX for FO logic, grounding to get a propositional formula is very straightforward. We can then simply call one of the many efficient modern SAT solvers for a solution. In the case of FO(ID) logic, the grounding will be a formula of propositional calculus with inductive definitions (PC(ID)). In this case, we would need to use a solver for PC(ID), e.g. [53, 48]. Essentially, we are reducing MX for FO to SAT, and MX for FO(ID) to satisfiability for PC(ID).

While there are other techniques that can be used other than grounding in order to make this declarative constraint programming approach of [?, ?] practical, since the solvers for propositional satisfiability are so efficient, an efficient grounding algorithm is a major part of this goal. While a naive approach would give us a polynomial time method for grounding (the problem description is fixed), which is expected as MX captures **NP**, the degree of this polynomial depends on the arity of the predicates, so something more efficient is needed. This same problem exists for grounding in ASP solvers as well, yet the practice of using domain predicates is used in some modern systems to overcome this problem. Since domain predicates are very much like "guards" in a logic program, it suggests that we could benefit by taking advantage of the guarded fragments (GF) [2] (and possibly their generalizations [24, 30, 6]) to devise an efficient grounding procedures for our framework, since MX is based on classical logic. We have indeed found these fragments to be useful, as they are central to the result of this thesis.

The result of this thesis is an efficient grounding algorithm for the MX framework which combines grounding with partial evaluation. In particular, when the FO(ID) formula is k-guarded [24], where all guards are specified by the instance structure, the algorithm runs in time $O(\ell^2 n^k)$, where ℓ is the length of the formula and n is the size of the structure. When k is small, this is a huge improvement over the naive time $O(n^\ell)$ approach. One reason that this result is important is that many search and decision problems that occur in practice can be written as k-guarded formulas, for small k, such that all guards are specified by the instance structure. This grounding algorithm is an extension of the model checking algorithm of Liu and Levesque [46], and is also inspired by the system Datalog LITE [21].

An overview of the thesis is as follows. In Chapter 2 we present related work, a set of implementations that are most closely related to the MX framework with this grounding procedure. Chapter 3 formally outlines the MX framework. We then give background information on the reasons behind the efficiency of our algorithm in Chapter 4, namely the notion of tree (hypertree) decompositions, and its connection to the k-guarded fragment of FO logic. Our contribution of the main result of this thesis begins in Chapter 5, with an outline of the mechanism of grounding for MX. First, we give a brief explanation of how we perform grounding and then we define the *extended relation*, an extension of the notion of relation from database theory. We then complete this mechanism by defining an algebra for this extended relation. Chapter 6 provides an algorithm for grounding FO formulas with expansion predicates over a structure. In the case where the FO formula is k-guarded such that all guards are specified by the instance structure, this algorithm runs in time $O(\ell^2 n^k)$, where ℓ and n are as above. Finally, in chapter 7, we provide the main algorithm of this thesis, namely the algorithm for grounding FO(ID) formulas with expansion predicates over a structure. In the case where the FO(ID) formula is k-guarded such that all guards are specified by the instance structure, this algorithm runs in time $O(\ell^2 n^k)$. This is done by trivially extending the algebra of Chapter 5 and the algorithm of Chapter 6 for inductive definitions. We then present conclusions followed by some future work.

Chapter 2

Related Work

In this chapter, we outline some specific declarative constraint programming approaches that employ grounding techniques that are similar to the one we present.

Since ASP is the latest approach to declarative constraint programming, a few of the related implementations are ASP systems. In particular, we mention the systems `Smodels` [55], `Cmodels-2` [43], and `dlv` [40].

The `Smodels` [55] system is an ASP system that extends normal logic programs with cardinality and weight constraints. It also supports arithmetic and function symbols. The set of *domain* predicates of an `Smodels` program is the maximally *stratifiable* subset of predicates of the program. A set of predicates (or rules) can be stratified if it can be ordered in such a way that for any of its predicates p and q, if p is definitionally dependent on q, then p is on at least as high a stratum as q. Note that all of these definitional dependencies must be positive, *i.e.*, p cannot depend on the negation of q. Note that if we create a graph for a stratified set of rules, where we take the predicates as nodes and we introduce a directed edge for each dependency, that this *positive dependency graph* would be a directed acyclic graph.

The idea here is that an `Smodels` program must be *domain-restricted*, that is, all variables of a rule must occur in some positive domain predicate in the body of this rule. This syntactic restriction guarantees that the interpretations of all predicates are subsumed by the intepretations of the (natural join of the) domain predicates, thus the set of domain predicates can be viewed as the *guards* of the program. This restriction guarantees that programs are decidable, even when function symbols are used [57]. While determining whether there exists a solution to an `Smodels` program is **2-EXP**-complete in the general case, imposing restrictions such as disallowing any function symbols or fixing the number of variables used can reduce the running-time significantly [57].

The system `Smodels` is composed of two independent components, `smodels` and `lparse`. Given an `Smodels` program, `lparse` first identifies the domain predicates and then computes the join of these predicates. It then uses this join to ground the rest of the program. The component `smodels` then runs on the ground program to find all of the solutions (or answer sets, in the context of ASP).

The form of grounding that `Smodels` employs is very similar to our approach, in that we both ground high-level programs into propositional programs and then rely on a lower-level solver to compute the solution. Furthermore, we both rely on the high-level programs having a certain structure, namely that the programs are *guarded* in some sense. The difference, however, is that `Smodels` is an ASP system so, while the notion of being domain-restricted is quite similar to being guarded [2], the grounding produced is an answer set program, not a SAT instance. Moreover, while imposing the same restrictions that we impose on the input to our grounding algorithm results in similar running times [57], `Smodels` considers less restricted cases as well, with the penalty of a much higher running time in these cases.

The system `Cmodels-2` [43] is another ASP system that is very similar to `Smodels`. It is similar to `Smodels` in that it uses `lparse` as a front end to ground the program. It also requires that the input be domain-restricted in the same sense as `Smodels`. `Cmodels-2` programs can have cardinality and weight constraints as well, however `Cmodels-2` uses methods to convert these into a set of nested rules with auxiliary variables [3]. The main idea behind `Cmodels-2` is that, while the set of answer sets of a general program is a subset of the *models of completion* of that program, the models of completion of a stratifiable program (its entire set of rules is stratifiable) are exactly the answer sets of this program. We explain the completion of a program below.

Recall that a (nested) logic program Π consists of a set of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_k, not\ A_{k+1}, \ldots, not\ A_m, not\ not\ A_{m+1}, \ldots, not\ not\ A_n$$

where A_0 is an atom (or the empty symbol), and $A_1, \ldots, A_n$ are atoms, where the list $A_1, \ldots, A_k$ is referred to as the *positive part* of the body [3]. If we assume that Π is stratifiable, the completion of this program can be formed as follows. First, in the body of every rule of Π replace each occurrence of *not* with $\neg$ and each comma with $\wedge$. Second, for every atom A make the list of all rules in Π with the head A

$$A \leftarrow Body_i$$

and form the equivalence

$$A \equiv \bigvee_i Body_i.$$

Third, for every constraint $A' \leftarrow Body$ in Π, where A' is the empty symbol, form the negation of its body

$$\neg Body.$$

Note that the completion of a program is a propositional formula, not a propositional answer set program, therefore SAT can be employed to find the models of this completion. When the program is stratifiable, SAT will return exactly the answer sets of the program. When the program is not stratifiable, however, `Cmodels-2` performs the computation of loop formulas [38] by a method quite similar to [45]. This computation of loop formulas makes the program stratifiable, and now it can be solved using program completion [3].

Grounding for `Cmodels-2` resembles our approach to grounding more than `Smodels` in that, not only does it ground high-level programs to lower-level programs by relying on some notion of guardedness, but the low-level program in the case of `Cmodels-2` is a SAT instance, not a propositional answer set program. However, since `Cmodels-2` is an ASP system, the method of grounding to a SAT instance is quite different. In fact, Lierler and Maratean [43] state that when using certain SAT solvers, they are used in close correlation with the `Cmodels` system for obtaining a solution, and not as black boxes. This technique allows for a more efficient solution, which is one of the advantages it has over Assat [45], a system that uses SAT as a black box. However, since the framework we consider is based on classical logic, using SAT as a black box is naturally the most efficient method, because of the clear separation of instance and problem description.

The system `dlv` handles disjunctive logic programs, that is rules with heads composed not of a single atom, but of a disjunction of atoms. This system also considers only *safe* programs, ones where for each rule, all variables of the rule are in some positive atom of that rule. The system `dlv` also considers weak constraints, which are constraints that need not be satisfied, however models that satisfy weak constraints are preferred over models that do not, all else being equal. Each weak constraint is associated with some measure of a "cost" for dissatisfying this constraint, where models that minimize an overall cost are preferred.

Disjunctive logic programming is more expressive than normal ($\vee$-free) logic programming, in that it captures all problems in the complexity class $\mathbf{NP}^{\mathbf{NP}}$, or $\mathbf{NP}$ with an $\mathbf{NP}$ oracle, while normal logic programming captures all problems in the complexity class $\mathbf{NP}$.

For example, the 3-coloring problem can be expressed in disjunctive logic using the following rules

$$\begin{aligned} r_1: &\quad \mathit{color}(X,r) \vee \mathit{color}(X,g) \vee \mathit{color}(X,b) \leftarrow \mathit{node}(X), \\ r_2: &\quad \leftarrow \mathit{edge}(X,Y), \mathit{color}(X,C), \mathit{color}(Y,C). \end{aligned}$$

Rule r_1 states that if X is a node, it is colored either red, green or blue, and rule r_2 states that it is never the case that two nodes that form an edge have the same color. The answer sets of this program correspond exactly to the 3-colorings of the graph ($\mathit{node}(\_)$, $\mathit{edge}(\_,\_)$). The fact that models are always minimal in ASP guarantees that, in every coloring, each node will have exactly one color [41].

The system `dlv` is considered state-of-the-art for disjunctive logic programming systems. The main aspect of `dlv` that can be compared directly to what we have considered so far is the methods that `dlv` uses for grounding its programs. Before running many of its other routines, `dlv` employs a routine called the *intelligent grounder* [40] or (IG), which uses two techniques which we explain below.

The first technique that IG uses is rule rewriting [16] inspired by the database optimization techniques of pushing selections and projections as far down the join tree as possible. Given a rule, one technique is to "project out" a variable that appears in a only a subset of its atoms. For example, given the rule r_1,

$$a(X) \vee b(Y) \leftarrow c(X,Z,W), d(Z,Y), e(Y,W)$$

we notice that variable Z appears only in atoms c and d of r_1's body. In this case, we can add the rule

$$f(X,Y,W) \leftarrow c(X,Z,W), d(Z,Y),$$

and substitute r_1 by a new rule r_1',

$$a(X) \vee b(Y) \leftarrow f(X,Y,W), e(Y,W).$$

Ground instances for r_1' are generated faster than for r_1, because Z is only involved in the join of c and d, not of the entire rule. The rewriting has the same effect as projecting out Z after the join of c and d is performed, as these are the only atoms that contain Z.

While this technique is not used in `Smodels` or `Cmodels` (although it may prove useful in these cases), it is akin to our technique for grounding. Our grounding algorithm takes advantage of the k-guarded fragment [24], that is, the input is a k-guarded formula. By the structure of k-guarded formulas, it has a form in which projections have been optimally pushed down the join tree (to the extent that no join involves more than k atoms) [46]. While these rewriting rules modify an instance to one where projections and selections are pushed down as far as possible, the input of our grounding algorithm already has this form.

The second technique used by the IG involves a set of methods for ordering joins in the (positive) body of its rules [41]. While the natural join of a set of atoms will have the same value, regardless of the order in which the join is performed, the intermediate sizes of the join, and therefore the time of computation of the join, can be drastically affected by this order. In this approach to chose an optimal join ordering, information about the atoms involved in the join is exploited. With this information, they use several different statistical measures to estimate the ordering that minimizes the join computation time.

Specifically, the IG employs a greedy algorithm, that at each step $i > 1$, a greedy choice is made to select the ith atom given that $i-1$ atoms have already been placed in an ordering. An atom A is chosen as the ith if A is minimal with respect to some selectivity criterion. They present three different selectivity criteria. The first favors the atom that has the most variables in common with the join of the $i-1$ atoms, and, of these, the one with associated relation of the smallest cardinality. The second criterion favors the atom A that leads to the smallest intermediate relation size (the one that minimizes the size of the join of the $i-1$ atoms with A). This intermediate relation size is estimated with the collected statistics of the atoms mentioned in the previous paragraph. The third criterion takes into account the size of the intermediate result and the variables that the the ith atom has in common with the join of the $i-1$ atoms. This third (combined) criterion is the one used for join-ordering in IG, as it has been shown through benchmarking, that it performs the best in most situations.

Test results show that this technique of join-ordering methods reduces the time of grounding quite significantly in the general case for `dlv` programs. This technique of ordering joins can be used in normal logic programming such as in the cases of `Smodels` and `Cmodels`. The grounding procedure of `dlv` has a clear edge over these methods in this regard. In fact, in one of the tests of [41], they compare the IG with the `lparse` grounder. In most cases, the IG performs better, especially in cases where the joins are quite large (where ordering is important). Since the results of our algorithm rely on the fact that most practical problems can be written in a k-guarded form for small k, we would not benefit from join ordering methods in grounding.

The techniques of rule rewriting [16] and join-ordering methods [41] make the IG a strong point of `dlv` with respect to other ASP systems. While it works well on general disjunctive logic programming problems, it even performs better than `Smodels` and `Cmodels` on some normal logic programming problems.

Since k-guarded sentences are those in which projections have been optimally pushed down the join tree, we can take advantage of this for efficient grounding. In this way, it resembles the rule rewriting techniques used in `dlv` in that they are inspired by database techniques of optimally pushing projections down the join tree. However, in our approach, we assume the input is already k-guarded, and can thus get an apriori bound (a function of k) on the running time, given this input.

In summary, the grounding approaches of the three systems mentioned above resemble our grounding approach, however a common way that they differ is that they are ASP systems, and not based on classical logic. The properties that all four approaches have in common is that they all require some notion of *safe* input, that is, all variables (or free variables) of a particular rule, or subformula, must be in some *positive* atom, guard or domain predicate. This second requirement, in fact, is very important to the efficiency of grounding (and of producing a solution) in all of these approaches.

Datalog LITE [21] is a model checker that runs in time linear in the program size and the input, that is, it has linear time combined complexity. Despite the fact that it performs model checking, and not model expansion, it is quite similar to our approach.

First consider Datalog LIT, the fragment of stratified datalog that is guarded [2][1], and therefore has linear time model checking. The reason a Datalog LIT program Π has linear time model checking is that the system can go through Π, stratum by stratum, and ground each rule r to a propositional Horn clause, of the form $h \vee \neg b_1 \vee \cdots \vee \neg b_n$, where h and the b_i are propositional atoms. Each rule r can be ground in linear time, because each rule is guarded (all variables take values within the extension of this guard). This yeilds a propositional Horn program that is linear in the size of Π. Since the satisfiability of a Horn clause can be determined in linear time, the entire process of model checking for Datalog

[1]or monadic, which we do not discuss as it is outside the scope of our interests

LIT is linear time.

Datalog LITE, an extension of Datalog LIT, differs only in that it may contain *generalized literals*; those of the form $\forall y_1 \cdots y_n(\alpha \supset \beta)$. While generalized literals are more powerful, the notion of stratification for these literals still has to hold, that is, α will have to be on a strictly lower stratum in the program than β. A generalized literal $L(\bar{y}) \leftarrow \forall \bar{x}(\alpha(\bar{x}, \bar{y}) \supset \beta(\bar{x}, \bar{y}))$, can be transformed to a set of regular literals by replacing it with the conjunction

$$\bigwedge_{\bar{c}:\ \alpha(\bar{c},\bar{d})\ \mathrm{true}} \beta(\bar{c}, \bar{d}),$$

for any tuple $\bar{d}$ interpreting $\bar{y}$. Since the number of tuples $\bar{d}$ is bound above by the input size, this replacement will only increase the program size by at most a linear factor. The routine for Datalog LITE, given program Π, performs this conversion for each generalized literal, to get a program Π' that is linear in the size of Π. It then calls Datalog LIT on Π', which does model checking in time linear in the size of Π', and therefore in time linear in the size of Π.

Datalog LITE most closely compares to our approach in the form of the input, guarded datalog. Instance and problem description are completely separate in Datalog LITE as well; Datalog LITE performs model checking for a program Π, given a structure $\mathcal{A}$. Thus, the method of grounding is quite similar as well, involving the extension of the language of the program Π by constant symbols for all domain elements of the structure $\mathcal{A}$. Datalog LITE has more in common with our work than the systems we have considered so far, and thus has inspired some of the approaches we take in our grounding algorithm. There are a number of key differences between Datalog LITE and our approach. First, Datalog LITE uses logic programming syntax, and grounding involves no partial evaluation. Finally, while they ground high-level specifications into lower-level specifications, it is for the much simpler task of model checking, not model expansion.

In the work of [54], the authors present an approach for converting an arbitrary FO formula into a propositional formula that is satisfiable iff the original FO formula is satisfiable. The main result of this paper is a method for propositionalizing FO formulas that are either in monadic first-order logic (MFOL), or exists-forall first-order logic (EAFOL). This method requires there to be a fixed number of constants, but does not rely on any other information, i.e., a structure or even a domain. For handling arbitrary FO formulas, a routine is provided for converting a FO formula to MFOL or EAFOL, however this relies on assuming that the domain of the formula is closed.

In order to get a compact propositionalization for the method of propositionalizing MFOL and EAFOL formulas, these methods rely on the results of [1]. Given a FO formula in MFOL or EAFOL, it is partitioned into sets of predicates/constants using the approach [1] based on tree-decomposition. Each partition can then be propositionalized independently, resulting in a propositional formula that is satisfiable iff the original MFOL or EAFOL formula is satisfiable. For classes of formulas of bounded treewidth, this propositionalization is polynomial in the size of the FO formula, that is, exponentially smaller than than the one obtained by the naive approach.

This approach of grounding is similar to ours in that it grounds a FO formula into a propositional formula so that SAT can be used to find a solution. It is also similar to our grounding approach in that it relies on the notion of tree-decomposition, however the techniques of [54] differ from our approach more than any of the systems we have mentioned above. Rather, the techniques rely on the partition-based reasoning of [1], rather than any notion of guardedness, or safe queries. They do rely on tree-decomposition, however, an idea that the notion of guardedness comes from, but they actually partition (or decompose) the formula with a greedy algorithm based on tree-decomposition. Our methods rely on the notion of tree-decomposition in that we restrict the input to being k-guarded, rather than performing any sort of decomposition. Finally, this approach concerns determining whether a problem is satisfiable, not of finding all satisfying assignments.

The system `NP-SPEC` is a system for specifying **NP** search problems in a Datalog-like syntax, where a compiler `SPEC2SAT` translates these high-level specifications into a propositional formula so that a SAT solver can generate solutions.

The system `NP-SPEC` has its own syntax, where a program consists of a database section and a specification section. A set of declaration forms are provided with `NP-SPEC`, where each specification has to use one of the forms. For example, the Hamilton path problem, given a graph $G = (V, E)$ could be specified using the `Permutation` declaration form [9]. That is, a Hamilton path can be specified as a permutation π of the vertices V such that $(\pi_i, \pi_{i+1}) \in E$ for all i s.t. $1 \leq i \leq |V| - 1$. This specification is then converted to an internal representation (based on simpler declaration forms native to `NP-SPEC`), and then finally to a propositional formula.

This syntax of `NP-SPEC` is restricted to being stratifiable (recursion through negation is not allowed), which guarantees specifications to be decidable, as in the ASP systems mentioned above. In addition, there are no cardinality or weight constraints, and no function symbols, which is necessary, since the grounding over the Herbrand Universe, unlike our system and Datalog LITE. Specifications of `NP-SPEC` are ground using the *minimal model* semantics [44]. Other than the semantics being slightly different than in our approach of grounding, their method of grounding is very similar to ours; they even combine grounding with partial evaluation.

The system `NP-SPEC` came out of the idea of specifying **NP** search problems in a highly declarative fashion. In fact, `NP-SPEC` captures the complexity class **NP**, by appealing to Fagin's theorem [17]. Both `NP-SPEC` and our approach combine grounding with partial evaluation. In contrast, `NP-SPEC` restricts input to a decidable fragment of programs, not the guarded fragment. Since there is no complexity analysis given for grounding for `NP-SPEC` (only experimental results), it is not clear how it differs from our approach in this vein. In addition, the fact that `NP-SPEC` relies on the minimal model semantics, and that grounding is over the Herbrand Universe set it apart from our approach, which relies on the semantics of classical logic, and grounds over the domain of the structure.

To conclude, all of these approaches have the same intent, which is to ground high-level specifications into low-level ones, so that a low-level solver can generate solutions. In all the systems but `Smodels` and `dlv`, this low level solver is SAT. In all of the systems but the approach of [1], some notion of a *safe* query or *guardedness* is used in order to guarantee decidability or efficiency. The system Datalog LITE resembles our approach most closely, in that there is a clear separation of instance and problem description. Furthermore, Datalog LITE considers guarded Datalog, and the problem instance is a structure. As such, Datalog LITE has inspired some of the techniques we use for an efficient grounding algorithm.

Chapter 3

Background: The Model Expansion Framework

The model expansion framework was first suggested by [49, 50], as an approach to purely declarative programming that is based as much as possible in classical logic. Here we fully present the model expansion framework for FO logic.

The language of the MX framework is simply FO logic. An FO language $\mathcal{L}$ contains of the set of *logical symbols*, which consists of

- parentheses: (,)
- connectives: $\wedge$, $\vee$, $\neg$, $\supset$
- variables: $x_1, x_2, \ldots$
- equality symbol: $=$

and the set of *parameters*, which consists of

- quantifiers: $\forall$, $\exists$
- predicate symbols: $P, Q, \ldots$ (for each positive integer n, we have some set (possibly empty) of n-ary predicate symbols)
- constant symbols: $c_1, c_2, \ldots$ (set can be empty)
- function symbols: $f_1, f_2, \ldots$ (set can be empty)

For example, if FO language $\mathcal{L}$ has variables x and y, and a binary relation L, one formula of this language $\mathcal{L}$ is $\forall x \exists y L(x,y) \vee \exists x \forall y \neg L(x,y)$. As an illustration, if the variables x and y range over all people, and $L(a,b)$ is interpreted to mean that "a likes b", this formula would say "Everybody likes someone, or there is someone who dislikes everyone".

Formally, to give meaning to the FO language $\mathcal{L}$, we define the class of $\mathcal{L}$-structures for this language, as well as an *object assignment*, *i.e.*, a mapping from the set of variables of ϕ (which we call V) to elements of the domain. An $\mathcal{L}$-structure $\mathcal{A}$ consists of

- A non-empty set A, called the domain (or universe) of $\mathcal{A}$. Variables of $\mathcal{L}$ range over this domain A
- For each n-ary function symbol $f \in \mathcal{L}$, a function $f^{\mathcal{A}}: A^n \to A$
- For each n-ary predicate symbol $P \in \mathcal{L}$, a n-ary relation $P^{\mathcal{A}} \subseteq A^n$
- If $\mathcal{L}$ contains $=$ then $=^{\mathcal{A}}$ must be the identity relation on A

An object assignment is simply the function

$$\varrho : V \to A$$

Now we can define $\mathcal{A} \models \phi[\varrho]$, that is, what it means for a structure $\mathcal{A}$ to satisfiy ϕ with object assignment ϱ. To do this, we formally define $\bar{\varrho}$, an extension of ϱ to all terms (a term is a constant or a constant with any number of functions applied to it) of ϕ (which we call T)

$$\bar{\varrho} : T \to A$$

and give the following definitions

Terms

- For each variable x, $\bar{\varrho}(x) = \varrho(x)$
- For each constant c, $\bar{\varrho}(c) = c^{\mathcal{A}}$
- If $t_1, \ldots, t_n$ are terms and f is an n-ary function,
  then $\bar{\varrho}(f(t_1, \ldots, t_n)) = f^{\mathcal{A}}(\bar{\varrho}(t_1), \ldots, \bar{\varrho}(t_n))$

Atomic Formulas

- $\mathcal{A} \models (t_1 = t_2)[\varrho]$ iff $\bar{\varrho}(t_1) = \bar{\varrho}(t_2)$
- For any n-ary predicate symbol P, $\mathcal{A} \models P(t_1, \ldots, t_n)[\varrho]$ iff $\langle \bar{\varrho}(t_1), \ldots, \bar{\varrho}(t_n) \rangle \in P^A$

Other Formulas

- $\mathcal{A} \models (\neg\phi)[\varrho]$ iff $\mathcal{A} \not\models \phi[\varrho]$
- $\mathcal{A} \models (\phi_1 \vee \phi_2)[\varrho]$ iff $\mathcal{A} \models \phi_1[\varrho]$ or $\mathcal{A} \models \phi_2[\varrho]$
- $\mathcal{A} \models (\phi_1 \wedge \phi_2)[\varrho]$ iff $\mathcal{A} \models \phi_1[\varrho]$ and $\mathcal{A} \models \phi_2[\varrho]$
- $\mathcal{A} \models (\forall x \phi)[\varrho]$ iff for every $d \in A$, we have $\mathcal{A} \models \phi[\varrho(x|d)]$, where

$$\varrho(x|d)(y) = \begin{cases} \varrho(y) & \text{if } y \neq x \\ d & \text{if } y = x \end{cases}$$

$\varrho(x|d)$ is like ϱ, except at the variable x it assumes the value d (which is the entire domain). Thus, $\forall$ means "for all elements of A". Note that all of the other symbols $\supset, \exists, \ldots$ can be defined by symbols in the above definitions; this gives the semantics of $\mathcal{L}$.

Since all FO languages $\mathcal{L}$ have the basic logical symbols, they tend only to differ in the *vocabulary*, that is, the sets of predicates, constants and functions. Therefore, we often refer to formulas ϕ as having a certain vocabulary, rather than being of a certain language. We use this convention, not mentioning languages again, and we use the symbols σ and ε for vocabularies. We also use the convention $\phi \in \sigma$ to mean that ϕ has vocabulary σ, or $vocab(\phi) = \sigma$. Furthermore, we only deal with relational vocabularies (no function symbols) from now on, both in explanation and algorithm implementation, for notational convenience. Note that since any n-ary function symbol can be replaced with an $(n+1)$-ary relation symbol in a way that the formula has the same meaning, that this is not a restriction. So in our case, we can describe a language simply as a relational vocabulary σ, and a σ-structure as $\mathcal{A} = (A; \sigma^{\mathcal{A}})$, where A is the domain of the structure, and $\sigma^{\mathcal{A}}$ is the interpretation of the relations of σ.

Let σ be a vocabulary, and $\mathcal{A}$ be a structure over σ, $\mathcal{A} = (A; \sigma^{\mathcal{A}})$. Let $\mathcal{B}$ be a structure over vocabulary $\sigma \cup \varepsilon$, the vocabulary σ extended with some vocabulary ε. Structure $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ is over the same domain as $\mathcal{A}$ and has the same interpretations of the relations of σ as $\mathcal{A}$, but also contains interpretations of the relations of ε. $\mathcal{B}$ is an *expansion* of $\mathcal{A}$. The MX framework for solving combinatorial search problems is given by

Definition 3.1 (the model expansion problem). *Given a FO sentence ϕ with vocabulary vocab(ϕ), and a finite structure $\mathcal{A}$ for vocabulary $\sigma \subseteq$ vocab(ϕ), is there a structure $\mathcal{B}$ which is an expansion of $\mathcal{A}$ to vocab(ϕ), such that $\mathcal{B} \models \phi$.*

To illustrate this framework and its suitability for solving combinatorial search problems, we consider the well-known **NP**-complete 3-coloring problem as a MX problem. The 3-coloring problem is, given a graph $G = (V, E)$ and three colors, is there a way to assign colors to the nodes such that for any $(a, b) \in E$, a and b cannot have the same color. Here we represent the problem instance, the graph G, with the structure $\mathcal{G} = (V; \sigma^{\mathcal{G}})$, where the domain of $\mathcal{G}$ is the set of vertices V, and $\sigma = \{E\}$ consists of the single binary (edge) relation. Here $E^{\mathcal{G}}$ (the interpretation of the relation E) is symmetric and irreflexive. We represent the problem description with the following formula ϕ

$$\forall x(R(x) \vee Y(x) \vee B(x)) \wedge$$

$$\forall x((R(x) \supset \neg(Y(x) \vee B(x))) \wedge (Y(x) \supset \neg(R(x) \vee B(x))) \wedge (B(x) \supset \neg(R(x) \vee Y(x)))) \wedge$$

$$\forall x \forall y(E(x, y) \supset (\neg(R(x) \wedge R(y)) \wedge \neg(Y(x) \wedge Y(y)) \wedge \neg(B(x) \wedge B(y))))$$

where the first line asserts that every node is colored. The second line asserts that no node has more than one color, and the third line asserts that no two nodes that share an edge have the same color. Thus, the formula is stated in such a way that every expansion of $\mathcal{G}$ that satisfies ϕ will represent a 3-coloring (if any exist). That is $R^{\mathcal{G}}$, $Y^{\mathcal{G}}$ and $B^{\mathcal{G}}$ will partition the set of vertices V, representing colors red, yellow and blue. The approach is clear; the structure represents the instance, and the formula describes the problem in a way that the set interpretations of the expansion predicates (expansion vocabulary $\varepsilon = \{R, Y, B\}$ in this case) is the witness of the solution. In this way we can formulate the decision problem as determining whether there exists an expansion of $\mathcal{G}$ that satisfies ϕ, and the assocated search problem as finding such an expansion.

From here on for model expansion problems, given formula ϕ and structure $\mathcal{A}$, we use the convention that σ represents the instance vocabulary, and ε represents the expansion vocabulary or *vocab*(ϕ) $\setminus \sigma$. Note that the model expansion problem where $\sigma =$ *vocab*(ϕ) reduces to the problem of *model checking*, *i.e.*, verifying if $\mathcal{A} \models \phi$. From here on we focus on the more interesting case where σ is a proper subset of *vocab*(ϕ). On the other hand, note that when $\sigma = \emptyset$, MX coincides with the spectrum problem. The spectrum of a sentence ϕ is the set $\{n \in \mathbb{N} \mid \phi$ has a finite model of size $n\}$. If $\sigma = \emptyset$ and *vocab*(ϕ) $=$

$\{R_1, \ldots, R_m\}$, the spectrum of ϕ can be alternatively viewed as finite models of the $\exists$SO sentence $\exists R_1, \ldots, \exists R_m \phi$. In particular, given an instance $(\phi, \mathcal{A})$ to the MX problem where $\sigma = \emptyset$ and the domain A of the structure $\mathcal{A}$ is of size n, we can determine if there is an expansion $\mathcal{B}$ of $\mathcal{A}$ such that $\mathcal{B} \models \phi$ by determining if n is in the spectrum of ϕ and vice versa.

Note that symbols in the expansion vocabulary ε behave as existentially quantified second order variables. So, in the case of FO logic, we have the same power as $\exists$SO over finite structures, that is, MX captures the complexity class **NP**. This can be proved using the same technique employed in Fagin's Theorem [17]. Formally, we define the following class of problems

Definition 3.2 (expressed by MX(σ, ϕ)). *A class of finite σ-structures K is expressed by MX(σ, ϕ) if for any σ-structure $\mathcal{A}$, $\mathcal{A} \in K$ iff there is an expansion $\mathcal{B}$ of $\mathcal{A}$ such that $\mathcal{B} \models \phi$.*

Assuming standard encoding of languages by classes of structures, and vice versa (see, e.g. [42]), we have the following re-casting of Fagin's Theorem. A class of finite σ-structures K is in **NP** iff for some FO formula ϕ, K is expressed by MX(σ, ϕ).

Some lower complexity classes can be captured similarily, applying results for various fragments of $\exists$SO [27, 39]. For example, FO universal Horn MX(σ, ϕ) expresses **P** over ordered structures. The Σ_k^p levels of the Polynomial Heirarchy (PH) are captured by Π_{k-1}^1MX(σ, ϕ). Note that MX does not naturally capture Π_k^p levels, and this is not just happenstance: If there are some σ and ϕ so that MX(σ, ϕ) is Π_k^p-complete, then the PH collapses to its k-th level. In particular, if there are σ and ϕ such that MX(σ, ϕ) is co-**NP**-complete, then **NP** = co-**NP**.

The MX framework allows us to consider the problems of model checking (MC), model expansion and finite satisfiability, all for the same logic. The complexity of MX lies between MC and finite satisfiability. In the case of FO logic, finite satisfiability is undecidable, MX is **NEXP**-complete, and MC is **PSPACE**-complete. Even in the case where $\sigma = \emptyset$, MX is decidable, as a finite structure is still provided. This finite structure has no relations, but the domain is finite. In fact, the case of the MX problem where $\sigma = \emptyset$ is equivalent to the result from [35] that the set $X \subset \mathbb{N}$ is a FO spectrum iff it is in **NEXP**. To prove **NEXP**-completeness for MX, it is a straightforward reduction from Bernays-Schönfinkel satisfiability or from the combined complexity of $\exists$SO over finite structures.

Although the complexity of MX in the general case is impractical, in many cases of the following parameterized version of the problem, it is practical.

Definition 3.3 (parameterized MX). *Fix an unrestricted FO formula ϕ and a vocabulary $\sigma \subset vocab(\phi)$. The problem $MX(\sigma, \phi)$ is: Given a finite structure $\mathcal{A}$ for the vocabulary σ, is there an expansion $\mathcal{B}$ of $\mathcal{A}$ to $vocab(\phi)$ such that $\mathcal{B} \models \phi$.*

Since ϕ represents problem description, and σ-structure $\mathcal{A}$ represents instance, if we fix ϕ, we can solve this particular problem for many instances. For example, once we have the formulation of the 3-coloring problem, we can solve this problem for many graphs G, given their corresponding structures $\mathcal{G}$. This class of problems describes the data complexity of MX, which is **NP**-complete, which is what we would expect, given that the 3-coloring problem is **NP**-complete, not **NEXP**-complete.

In regards to grounding, in the case where the formula ϕ is fixed, any naive grounding algorithm will produce a propositionalization in time $O(n^{\ell})$, where ℓ is the length ($|\phi|$) of the formula, and n is the size of structure $\mathcal{A}$. Note that this is polynomial in n because ϕ is fixed (which is expected, as grounding is a way to solve **NP**-complete problems by reducing MX to SAT). However, the algorithm we present produces a grounding in time $O(\ell^2 n^k)$ where ℓ and n are as above. This polynomial can be much smaller than the above $O(n^{\ell})$ when k is much smaller than ℓ. In practice, many problems can be formulated as model expansion problems where the formula can be written as a k-guarded formula for small k. This is why the results for this particular fragment of FO logic are useful from a practical perspective.

Chapter 4

Background: The k-Guarded Fragment

The guarded fragment GF of FO was introduced by Andréka *et al.* [2]. Here any existentially quantified subformula ϕ must be conjoined with a guard, *i.e.*, an atomic formula over all free variables of ϕ. Gottlob *et al.* [24] extended GF to the k-guarded fragment GF_k of FO (which we refer to as GF_k from here on, mentioning explicitly when referring to the k-guarded fragment of some other logic), where the conjunction of up to k atoms may act as a guard, and proved that k-guarded sentences can be evaluated in time $O(\ell^2 n^k)$ where ℓ is the size of the formula, and n is the size of the structure. The proof is by transforming GF_k sentences into k-guarded Non-Recursive Stratified Datalog (NRSD) programs. Later, in [46], the authors show that model checking for a formula $\phi \in \text{GF}_k$ can be carried out in time $O(\ell n^k)$, where ℓ is the length of the formula, and n is the size of the (arbitrary) structure [46] (*i.e.*, the combined time complexity is polynomial). It is the structural properties of k-guarded formulas that gives rise to these results, *i.e.*, the polynomial time combined complexity of model checking for formulas in GF_k, over the general **PSPACE**-complete time complexity for model checking. We show in this thesis that taking advantage of the structural properties of k-guarded formulas is also beneficial in the case of coming up with an efficient grounding algorithm. The discovery of this fragment of FO logic has roots in structural graph theory, namely the notion of *tree decomposition* [7] and its generalizations.

A tree decomposition is a mapping from a graph to a tree. More specifically,

Definition 4.1 (tree decomposition). *Given graph $G = (V, E)$ a* tree decomposition *of*

G, TD(G), is a pair $(\mathcal{T}, \chi)$ where $\mathcal{T} = (N, T)$ is a tree, and $\chi : N \to 2^V$ is a mapping from vertices of $\mathcal{T}$ to sets of vertices of G. This pair $(\mathcal{T}, \chi)$ must meet the following constraints:

1. $\forall v \in V, \exists n \in N$ *s.t.* $v \in \chi(n)$
2. $\forall (u, v) \in E, \exists n \in N$ *s.t.* $u \in \chi(n) \wedge v \in \chi(n)$
3. $\forall v \in V, \{n \in N \mid v \in \chi(n)\}$ *induces a (connected) subtree of* $\mathcal{T}$

In this definition, the first rule states that every vertex of G is associated with at least one vertex of $\mathcal{T}$. The second rule states that each pair of vertices of every edge of G is associated with at least one vertex of $\mathcal{T}$. The last rule states that the set of vertices of $\mathcal{T}$ that each vertex of G associates with induces a connected subtree of $\mathcal{T}$. The *tree-width* of a given tree decomposition TD of G is $\max_{n \in N} |\chi(n)| - 1$, and the tree-width of a graph G is the minimum width of all tree decompositions of G.

Intuitively, a tree decomposition is an arrangement of the graph as a tree, and the tree-width of a graph is a measure of how well the graph "fits" to a tree (the better the fit, the closer to 1, from infinity, the tree-width is). If a graph G that has bounded tree-width, that is the tree-width of G is some constant k, this fact can be recognized in linear $(O(n))$ time. Given this, a width k tree decomposition of G can be built in $O(n)$-time. Since many **NP** graph properties become polynomial-time testable (Maximum Independent Set, 3-Colorability, etc.) when the graph is a tree, this is also the case when the graph has bounded tree-width. This is because, given graph G of tree-width at most k, we can both recognize this fact, and construct a tree decomposition TD(G) $= (\chi, \mathcal{T})$ of G in polynomial time. Given this, we can then modify the property test to work for the tree $\mathcal{T}$ in polynomial time, because each node of the tree contains only a constant amount of information.

Since many problems that can be represented as graphs have small tree-width, this notion has had much success in practice. However many problems are more faithfully represented in terms of hypergraphs, rather than graphs. As such, it would be useful if such a notion of decomposition existed for hypergraphs, so that properties of low "width" hypergraphs could be tested efficiently. A hypergraph is a generalization of the notion of graph. While a graph G is a set of vertices and a set of pairs of vertices (or edges), a hypergraph H is a set of vertices and set of subsets of vertices (or hyperedges). That is, hypergraphs can represent objects related to each other by not just binary relations, but relations of any arity. For example, a FO formula

$$\exists vwxz(S(v,x,y) \wedge R(x,y,z) \wedge T(z,w)) \tag{4.1}$$

can be viewed as a hypergraph (Figure 4.1). In this case, the vertices represent the variables, and the atoms are relations (hyperedges) over these vertices. If there is some valuation for the variables that satisfies formula 4.1, the variable x will have to take the same value at all places (in S and in R), so we can represent x as a single vertex. Furthermore each atom relates the variables within it by requiring that the tuple of valuations actually satisfy the given atom. Thus, given that each atom is not only binary in general, with hypergraphs is the best way to represent these entities.

An important property test for formulas (which have a hypergraph structure) is the model checking problem. The model checking problem is, given a σ-formula ϕ and a σ-structure $\mathcal{A}$, does $\mathcal{A} \models \phi$. The length ℓ of the formula ϕ is the number of atoms in the formula (or the number of variables; from an asymptotic viewpoint, they are the same). The size n of the structure $\mathcal{A}$ is the size of any reasonable encoding of this structure. An example of a reasonable encoding of a structure is an encoding of the domain of the structure followed by an encoding of each tuple of each relation interpreted by the structure. A detailed example of an encoding of a structure can be found in [18], thus n can be used in place of the size of domain A of $\mathcal{A}$, or the size of the interpretation (table of tuples) of any relation of $\mathcal{A}$. Given formula 4.1, and structure $\mathcal{A}_1 = \{S^{\mathcal{A}_1} = \{(2,6,6),(3,4,5),(5,2,3)\}, R^{\mathcal{A}_1} = \{(2,7,8),(4,5,6)\}, T^{\mathcal{A}_1} = \{(6,1),(6,3),(8,2)\}\}$, $\mathcal{A}_1$ satisfies formula 4.1 because the tuples (3,4,5), (4,5,6), (6,1) satisfy the set of atoms S, R and T respectively, and are consistent with the variables.

From a computation viewpoint, for formula ϕ and structure $\mathcal{A}$, checking that $\mathcal{A} \models \phi$ involves computing the *answer* to ϕ w.r.t. $\mathcal{A}$. To describe the answer to a formula w.r.t. a structure, we introduce the notion of relation [18] from relational database theory. An X-relation $\mathcal{R}$ is a set of mappings from the set of free variables X of the formula, to the set of elements A of the domain of the structure, or $\mathcal{R} = \{\gamma : X \to A\}$. The answer to an atomic formula w.r.t. a structure is the X-relation that is the set of mappings of the free variables of the atom to the tuples of the interpretation of the atom in the structure. For example, the answer to atomic formula $S(v,x,y)$ w.r.t. $\mathcal{A}_1$ is $\{\gamma_1 : \{v,x,y\} \to \{2,6,6\}, \gamma_2 : \{v,x,y\} \to \{3,4,5\}, \gamma_3 : \{v,x,y\} \to \{5,2,3\}\}$ or, the following table on the left

v	x	y
2	6	6
3	4	5
5	2	3

v	x
2	6
3	4
5	2

The Y-*projection* $\pi_Y(\mathcal{R})$ of X-relation $\mathcal{R}$, where $Y \subseteq X$ is the Y-relation $\{\gamma|_Y \mid \gamma \in \mathcal{R}\}$, or the tuples of $\mathcal{R}$ restricted to the set Y. For example, $\pi_{\{v,x\}}(S(\mathcal{A}_1))$ is the above table on the right. Note that given that the size $||\mathcal{R}||$ of X-relation $\mathcal{R}$ is its arity times the number of tuples (in the case of $\{v,x,y\}$-relation $S(\mathcal{A}_1)$, $||S(\mathcal{A}_1)||$ is $3 \cdot 3 = 9$). Given this, the Y-projection of $\mathcal{R}$ involves passing through the tuples once and discarding the columns of X that fall outside of Y, resulting in a table that is smaller or equal in size to $\mathcal{R}$. Thus Y-projection is an $O(||\mathcal{R}||)$-time operation, or a linear time operation.

The *join* $\mathcal{R} \bowtie \mathcal{S}$ of X-relation $\mathcal{R}$ and Y-relation $\mathcal{S}$ is the $X \cup Y$-relation $\{\gamma : X \cup Y \to A \mid \gamma|_X \in \mathcal{R}, \gamma|_Y \in \mathcal{S}\}$. For example, given $S(\mathcal{A}_1)$ and the fact that $R(\mathcal{A}_1)$ is the following table on the left

x	y	z
2	7	8
4	5	6

v	x	y	z
3	4	5	6

$S(\mathcal{A}_1) \bowtie R(\mathcal{A}_1)$ is the above table on the right. When X and Y are disjoint sets, the join of X-relation $\mathcal{R}$ and Y-relation $\mathcal{S}$ will be of size $||\mathcal{R}|| \cdot ||\mathcal{S}||$, as every tuple of $\mathcal{R}$ is matched with every tuple of $\mathcal{S}$. Since this is the worst case scenario, the join of $\mathcal{R}$ and $\mathcal{S}$ is a an $O(||\mathcal{R}|| \cdot ||\mathcal{S}||)$-time operation. A special case of the join operation is to join X-relation $\mathcal{R}$ and Y-relation $\mathcal{S}$ when $Y \subseteq X$. In this case, we simply have to sort the tables $\mathcal{R}$ and $\mathcal{S}$ according to a lexicographic ordering on A using the linear-time bucketsort (as the domain A is fixed) [18]. After the sort, the join is the same as merging two sorted lists, as X subsumes Y. Thus, in this case, join is an $O(||\mathcal{R}|| + ||\mathcal{S}||)$-time operation. The optimal way to do model checking, in this case, is to rewrite the formula so that as many joins as possible are linear time joins.

On the logical level, projections correspond to existential quantifications, and joins correspond to conjunctions. Until we get to the discussion on the k-guarded fragment of full FO logic, we only discuss the fragment $\exists\text{FO}_{\wedge,+}$ of FO logic (FO logic restricted to conjunction and existential quantification), as this is all that is needed to describe the *structure* of k-guarded formulas from a graph theoretic (or hypergraph theoretic) standpoint. This is why

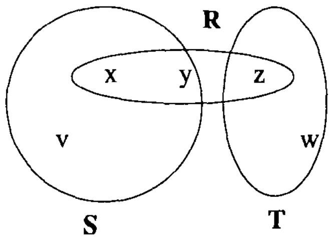

Figure 4.1: formula 4.1 as a hypergraph

we only mention projection and join, leaving the definitions of the operations that pertain to disjunction and negation until the discussion of full FO logic. So the answer to formula 4.1 w.r.t. $\mathcal{A}_1$ is $\pi_{\{y\}}(S(\mathcal{A}_1) \bowtie R(\mathcal{A}_1) \bowtie T(\mathcal{A}_1))$, that is, we join the three conjoined relations, and then we project to the set $\{y\}$, as all variables but y are existentially quantified and thus go out of scope. This computation is as follows: compute $S(\mathcal{A}_1) \bowtie R(\mathcal{A}_1)$ resulting in the table above, then we compute $S(\mathcal{A}_1) \bowtie R(\mathcal{A}_1) \bowtie T(\mathcal{A}_1)$ resulting in the table on the left

v	x	y	z	w
3	4	5	6	1
3	4	5	6	3

y
5

This is then projected onto the set $\{y\}$ to get the above table on the right. Since this table is not empty, there is a valuation to y of formula 4.1 from structure $\mathcal{A}_1$, such that this formula is satisfied (*i.e.*, the structure satisfies the formula under some object assignment). In the general case, for formula 4.1, model checking involved three join operations, and a projection at the end, thus the total time is $O(n^3)$, as each atomic relation is of size $O(n)$ and three were joined together to get intermediate relation of size $O(n^3)$. Then a linear-time projection was applied to this intermediate relation for a final relation of size $O(n^3)$. In the general case for positive conjunctive formulas, the time is $O(n^\ell)$, since there are $O(\ell)$ atoms in the formula.

A more clever way to do this, however, would be to rewrite the formula 4.1 into

$$\exists xz(\exists v(S(v,x,y)) \wedge R(x,y,z) \wedge \exists w(T(z,w))) \tag{4.2}$$

Note that formula 4.2 is semantically equivalent to formula 4.1 (has the same meaning, *i.e.*, has the same models). This rewriting implies that operations will now be done in the order

$\pi_{\{y\}}((\pi_{\{x,y\}}(S(\mathcal{A}_1))) \bowtie R(\mathcal{A}_1) \bowtie (\pi_{\{z\}}(T(\mathcal{A}_1))))$. This computation is as follows: compute $\pi_{\{x,y\}}(S(\mathcal{A}_1))$ resulting in the following table on the left

x	y
2	3
4	5
6	6

x	y	z
4	5	6

Then compute $(\pi_{\{x,y\}}(S(\mathcal{A}_1))) \bowtie R(\mathcal{A}_1)$ resulting in the above table on the right. This is an instance of the special case of join, since it is the join of a $\{v, x, y\}$-relation and a $\{x, y\}$-relation. Next is to compute the join of the above relation with $\pi_{\{z\}}(T(\mathcal{A}_1))$. Since $\pi_{\{z\}}(T(\mathcal{A}_1))$ is the following table on the left

z
6
8

x	y	z
4	5	6

y
5

The join $(\pi_{\{x,y\}}(S(\mathcal{A}_1))) \bowtie R(\mathcal{A}_1) \bowtie (\pi_{\{z\}}(T(\mathcal{A}_1)))$ results in the above table in the center. This is also an instance of the special case of join as it is a join of a $\{x, y, z\}$-relation and a $\{z\}$-relation. Last is to project this relation onto the set $\{y\}$, resulting in the above table on the right. Note that this is same table we got in the case of formula 4.1, which which is what we would expect, since it and formula 4.2 are semantically equivalent.

This clever rearrangement involved moving projections inward as far as possible in order to keep intermediate joins small instead of joining large relations, which could produce a huge relation, and then projecting at the end. In fact, in the above case, projections could be moved in to the extent that each intermediate join remained of size $O(n)$. In general, for formula 4.2, this can be seen by the fact that we started with relations all of size $O(n)$, and that every operation was a linear-time operation. This means that the answer to formula 4.2 w.r.t. $\mathcal{A}_1$ was computed in time $O(n)$, a big improvement over $O(n^3)$. In general, when computing formulas where projections can be moved inwards to the point that every join can be done in linear time (*i.e.*, is a special case join), the time to compute the answer to the formula w.r.t. the structure is $O(\ell n)$. This is because we start with relations all of size $O(n)$ and then there are $O(\ell)$ linear operations, therefore all intermediate relations remain of size $O(n)$.

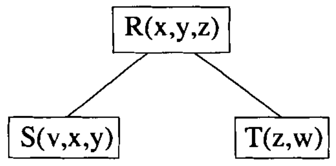

Figure 4.2: join tree of formula 4.1

The reason that the above formula 4.1 could be rewritten as formula 4.2 such that the combined complexity of its evaluation w.r.t. a structure is linear is because formula 4.1 has a *join tree* [22]. A join tree for a positive conjunctive formula ϕ is an arrangement of its atoms in a tree such that for every variable of the formula, the set of atoms containing this variable induces a (connected) subtree of this join tree. Let V be the set of variables of ϕ and At be the set of atoms of ϕ. Since atoms contain variables, we use the notation $v \in \alpha$ to denote that variable v is contained in atom α, and we use $var(\alpha)$ to denote the set of variables of atom α. Formally,

Definition 4.2 (join tree). *A* join tree *for positive conjunctive FO formula ϕ is a triple $(\mathcal{T}, \chi, \lambda)$ where $\mathcal{T} = (N, T)$ is a tree, $\chi : N \longrightarrow 2^V$ is a mapping from vertices of $\mathcal{T}$ to sets of variables of ϕ, and $\lambda : N \longrightarrow At$ is a mapping from vertices of $\mathcal{T}$ to atoms of ϕ. This triple $(\mathcal{T}, \chi, \lambda)$ must meet the following constraints*

1. $\forall \alpha \in At, \exists n \in N$ *s.t.* $var(\alpha) \subseteq \chi(n)$
2. $\forall \alpha \in At, \exists n \in N$ *s.t.* $\lambda(n) = \alpha$
3. $\forall v \in V, \{n \in N \mid v \in \chi(n)\}$ *induces a (connected) subtree of* $\mathcal{T}$
4. $\forall n \in N, \chi(n) = var(\lambda(n))$

In this definition the first rule states that every atom is *covered* by some vertex in tree $\mathcal{T}$, that is, given an atom of the formula ϕ, there is some vertex n of $\mathcal{T}$ such that all the variables of that atom are associated with n. The second rule states that every atom itself is associated with some vertex in tree $\mathcal{T}$. The third rule states that the set of vertices of $\mathcal{T}$ that each variable of the formula ϕ associates with induces a connected subtree of $\mathcal{T}$. The last rule states that for each vertex n of $\mathcal{T}$, the variables associated with n are exactly the variables of the atom associated with n. Figure 4.2 shows the join tree for the formula 4.1.

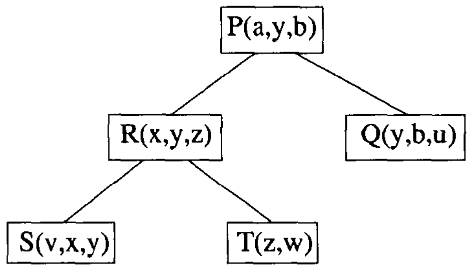

Figure 4.3: join tree of formula 4.3

Every formula ϕ that has a join tree can be evaluated in $O(\ell n)$ time. This is because the atoms of ϕ can be arranged in a way that join operations are done from the leaves of the tree to some (arbitrarily chosen) root of the tree. In this computation, for any pair of atoms α and β of ϕ that are joined by an edge in the tree, where α is closer to the root, we only need to consider the variables that β has in common with α in the join of α and β. That is, all of the projections (or existential quantifications) for the variables in the subtree rooted at β that have nothing in common with α can be pushed below α into this subtree, which means that all joins are special case (linear time) joins. For example, the positive conjunctive FO formula

$$\exists abuvwxyz(S(v,x,y) \wedge R(x,y,z) \wedge T(z,w) \wedge P(a,y,b) \wedge Q(y,b,u)) \tag{4.3}$$

has the join tree shown in Figure 4.3 (notice that it has the join tree of formula 4.1 as a subtree). This means that the formula can be rewritten into the positive conjunctive FO formula

$$\exists ayb(\exists xz(\exists v(S(v,x,y)) \wedge R(x,y,z) \wedge \exists w(T(z,w))) \wedge P(a,y,b) \wedge \exists u(Q(y,b,u))) \tag{4.4}$$

such that it can be evaluated with linear-time combined complexity.

Note the similarity of the definition of join tree for formulas to the definition of tree decomposition for graphs. In fact a positive conjunctive FO formula ϕ has a join tree iff its associated hypergraph is *acyclic.* Given a hypergraph $H = (V, E)$, the GYO-reduct [22] GYO(H) is the hypergraph obtained by applying the following rules as long as possible

1. remove hyperedges that are empty or contained in other hyperedges

2. remove vertices that appear in at most one hyperedge

If GYO$(H) = (\emptyset, \emptyset)$ then H is acyclic. So the definition of a join tree is a decomposition of the acyclic *hypergraph* into a tree (where we use vertices of H in place of variables of ϕ and hyperedges of H in place of atoms of ϕ, which we do interchangeably throughout, due to this equivalence).

So just as certain properties of acyclic graphs can be tested in polynomial (actually linear) time, so is the case for acyclic hypergraphs. While we will only mention satisfiability in the context of model checking, as it suits our purposes, there are many other examples (CSP, Homomorphism, etc.). The obvious question is whether the notion of acyclicity for hypergraphs can be generalized. In other words, is there a measure of acyclicity for hypergraphs, *i.e.*, a notion of decomposition and bounded width for hypergraphs as there is for graphs? There is such a notion, as we show in the remainder of this chapter. This notion is very useful, as it is the basis for the efficiency of our grounding algorithm

The first hypergraph decomposition notion that generalizes acyclicity is the notion of *query decomposition*, proposed in [10]. Like a join tree, a query decomposition of a hypergraph (formula) is an arrangement of its hyperedges (atoms) in a tree. However, with query decomposition, more than one hyperedge may be associated with each vertex of the underlying tree, where the connectedness condition still holds for each vertex (variable) in the hypergraph. Given hypergraph $H = (V, E)$, we use the notation $v \in e$ to denote that vertex v is contained in hyperedge e, and given an edge $e \in E$, we use $vertices(e)$ to denote the set of vertices of this hyperedge. Given a set of hyperedges $E' \subseteq E$, we use $vertices(E')$ to denote the set of vertices of this set of hyperedges, or $\bigcup_{e \in E'} vertices(e)$.

Definition 4.3 (query decomposition). *A* query decomposition *of hypergraph $H = (V, E)$ is a triple $(\mathcal{T}, \chi, \lambda)$ where $\mathcal{T} = (N, T)$ is a tree, $\chi : N \rightarrow 2^V$ is a mapping from vertices of $\mathcal{T}$ to sets of vertices of H, and $\lambda : N \rightarrow 2^E$ is a mapping from vertices of $\mathcal{T}$ to* sets of *hyperedges of H. This triple $(\mathcal{T}, \chi, \lambda)$ must meet the following constraints*

1. $\forall e \in E, \exists n \in N$ *s.t.* $vertices(E) \subseteq \chi(n)$

2. $\forall e \in E, \exists n \in N$ *s.t.* $e \subseteq \lambda(n)$

3. $\forall v \in V, \{n \in N \mid v \in \chi(n)\}$ *induces a (connected) subtree of* $\mathcal{T}$

4. $\forall n \in N, \chi(n) = vertices(\lambda(n))$

In this definition the first rule states that every hyperedge is *covered* by some vertex in tree $\mathcal{T}$, that is, given a hyperedge of H, there is some vertex n of $\mathcal{T}$ such that all the vertices of this hyperedge are associated with n. The second rule states that every hyperedge itself is associated with some vertex in tree $\mathcal{T}$. The third rule states that the set of vertices of $\mathcal{T}$ that each vertex of H associates with induces a connected subtree of $\mathcal{T}$. The last rule states that for every vertex n of $\mathcal{T}$, the set of vertices associated with n are exactly the set of vertices of the set of hyperedges associated with n. The *query-width* of a given query decomposition is $\max_{n \in N} |\lambda(n)|$, and the query-width of a hypergraph H is the minimum query-width of all query decompositions of H. The query-width of a formula is the query-width of its associated hypergraph. Note that the only difference in this definition from that of a join tree is in the second rule, that is, more than one hyperedge may be associated with a vertex of $\mathcal{T}$. Because of this, not just acyclic hypergraphs have query decompositions (all hypergraphs do), however the ones that are "close" to being acyclic will have low query-width. In fact, acyclic hypergraphs have query-width one, that is, each node of $\mathcal{T}$ has exactly one hyperedge associated with it, and thus query decomposition coincides with join tree on acyclic hypergraphs.

As an example, positive conjunctive FO formula

$$\begin{aligned}\exists sxx'cfsyy'c'f'zj(A(s,x,x',c,f) \wedge B(s,y,y'c'f') \wedge C(c,c',z) \wedge D(x,z) \wedge E(y,z) \\ \wedge F(f,f',z') \wedge G(x',z') \wedge H(y',z') \wedge J(j,x,y,x',y'))\end{aligned} \tag{4.5}$$

has the query decomposition shown in Figure 4.4. This is a minimal query decomposition for this formula, and thus formula 4.5 has query-width 3. Note that here, regarding model checking of a structure $\mathcal{A}$ of size n with ϕ of length ℓ, since the largest number of atoms associated with any vertex is 3, joining the atoms at any vertex of $\mathcal{T}$ will take time $O(n^3)$. After the joins at each vertex, we have a join tree of size $O(\ell)$ such that all relations are of size $O(n^3)$, thus formula 4.5 can be evaluated in $O(\ell n^3)$-time. So while acyclic formulas can be evaluated in $O(\ell n)$-time, formulas of query-width k can be evaluated in $O(\ell n^k)$-time.

The number of atoms, not the number of variables, associated with each vertex of the tree $\mathcal{T}$ determines the complexity of model checking for formula ϕ, since there are interpretations for atoms (and not variables) in a structure. Note that many of the other hypergraph properties are efficiently testable when we restrict vertices of $\mathcal{T}$ in terms of hyperedges of H

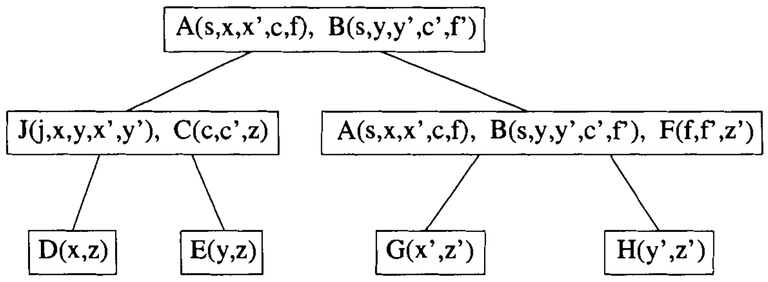
A(s,x,x',c,f), B(s,y,y',c',f')
J(j,x,y,x',y'), C(c,c',z)
A(s,x,x',c,f), B(s,y,y',c',f'), F(f,f',z')
D(x,z)
E(y,z)
G(x',z')
H(y',z')

Figure 4.4: query decomposition of formula 4.5

instead of vertices of H (CSP, Homomorphism, etc.). This is why query decomposition considers the number of atoms associated with a vertex of $\mathcal{T}$, and not the number of variables. There is a notion of tree decomposition for hypergraphs as well. The *Gaifmann graph* (or *primal graph*) of a hypergraph $H = (V, E)$ is the graph $G = (V, E^G)$, where $E^G = \{(u, v) \mid u$ and v are both in some hyperedge of $E\}$. In other words, it is the graph obtained by taking the vertices V of H and introducing a clique on any set of vertices that are related by a hyperedge in H. A tree decomposition of a hypergraph $H = (V, E)$ is a tree decomposition of its primal graph, and the tree-width of H is the tree-width of its primal graph. While this notion is useful, *i.e.*, model checking for a formula of bounded tree-width has polynomial time combined complexity, the converse is not always true. That is, if model checking for a class of formulas has polynomial time combined complexity, this need not be the class of formulas of bounded treewidth. For example, an acyclic formula with an atom of arity k has tree-width $k - 1$.

Query decomposition for hypergraphs, while a natural extension of the definition of tree decomposition for graphs does not serve quite the same function as tree decomposition. For one reason, it is **NP**-complete to decide whether or not a hypergraph has query-width 4, *i.e.*, hypergraphs of bounded query-width are not recognizable in polynomial time. Furthermore, query decompositions of hypergraphs are not optimal. Evidence of this fact is that formula 4.5 can be replaced with a semantically equivalent formula that has the (minimal) query decomposition of size 2 shown in Figure 4.5. From this query decomposition, it should be clear how this formula has been constructed from formula 4.5, it is a modification of formula 4.5 by introducing the six new variables $\alpha, \beta, \gamma, \delta, \epsilon, \gamma$ and reusing atom J. Since all of the original atoms (constraints) are there and these new variables appear in no other atom, *i.e.*, no new constraints are introduced, this new formula is semantically equivalent

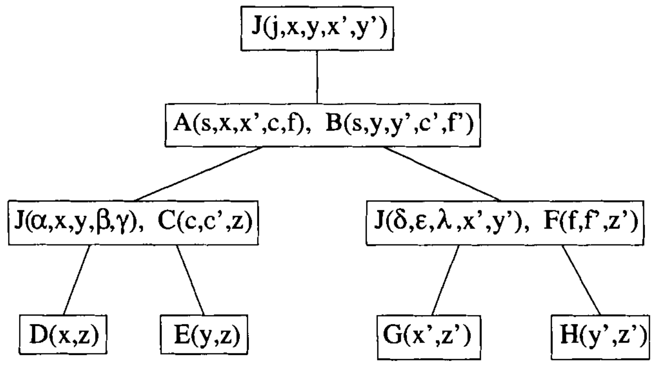

Figure 4.5: query decomposition of formula equivalent to formula 4.5

to formula 4.5. This means that essentially, we can do model checking for formula 4.5 in $O(\ell n^2)$-time as opposed to $O(\ell n^3)$-time. Fortunately, it turns out that the high complexity of determining bounded query-width is not, as one might expect, the price for the generality of the concept. Rather, it is due to some peculiarity in its definition [23]. Next we present a notion that is just as general and does not suffer from such problems.

The notion of *hypertree decomposition* was first proposed in [24], and is a notion that generalizes tree decomposition, acyclicity (or the concept of a join tree), query decomposition and is tractable. In fact formula 4.5 has a minimal hypertree decomposition of hypertree-width 2, which is why it can be evaluated w.r.t. model checking in $O(\ell n^2)$-time. Formally,

Definition 4.4 (hypertree decomposition). *given hypergraph $H = (V, E)$, a* hypertree decomposition *for H is a triple ($\mathcal{T}$, χ, λ) where $\mathcal{T} = (N, T)$ is a* rooted *tree*, $\chi : N \longrightarrow 2^V$ *is a mapping from vertices of $\mathcal{T}$ to sets of vertices of H, and $\lambda : N \longrightarrow 2^E$ is a mapping from vertices of $\mathcal{T}$ to sets of hyperedges of H. If $\mathcal{T}' = (N', T')$ is a subtree of $\mathcal{T}$, we define $\chi(\mathcal{T}') = \bigcup_{n \in N'} \chi(n)$. For any $n \in N$, $\mathcal{T}_n$ denotes the subtree of $\mathcal{T}$ rooted at n. This triple ($\mathcal{T}$, χ, λ) must meet the following constraints:*

1. $\forall e \in E, \exists n \in N$ *s.t.* $\mathit{vertices}(e) \subseteq \chi(n)$
2. $\forall e \in E, \exists n \in N$ *s.t.* $e \subseteq \lambda(n)$
3. $\forall v \in V, \{n \in N \mid v \in \chi(n)\}$ *induces a (connected) subtree of* $\mathcal{T}$
4. $\forall n \in N, \chi(n) \subseteq \mathit{vertices}(\lambda(n))$
5. $\forall n \in N, \mathit{vertices}(\lambda(n)) \cap \chi(\mathcal{T}_n) \subseteq \chi(n)$

In this definition the first rule states that every hyperedge is *covered* by some vertex in tree $\mathcal{T}$. The second rule states that every hyperedge itself is associated with some vertex in tree $\mathcal{T}$. The third rule states that the set of vertices of $\mathcal{T}$ that any vertex of the H associates with induces a connected subtree of $\mathcal{T}$. It is these last two rules that sets hypertree decompositions apart from query decompositions. The fourth rule states that for every vertex n of $\mathcal{T}$, the set of vertices associated with n can be a subset of the set vertices of the hyperedges associated with n, instead of the sets having to be equal. This relaxation allows hyperedges to be "reused" in vertices of $\mathcal{T}$ with only a subset of their vertices, in order to achieve a smaller width. The fifth rule states that for every vertex n of $\mathcal{T}$, the set vertices associated with the subtree rooted at n that is in common with the set of vertices of the hyperedges associated with n must be a subset of the set of vertices associated with n. In other words, for a given vertex n of $\mathcal{T}$, a vertex v of V can be a member of $\mathit{vertices}(\lambda(n))$, but not a member of $\chi(n)$, *i.e.*, it is a member of $\mathit{vertices}(\lambda(n)) \setminus \chi(n)$, as long as it is *not a member* of any $\chi(n')$ for any vertex n' of $\mathcal{T}$ in the subtree $\mathcal{T}_n$. Again the *hypertree-width* of a given hypertree decomposition is $\max_{n \in N} |\lambda(n)|$, and the hypertree-width of a hypergraph H is the minimum hypertree-width of all hypertree decompositions of H.

While the definition of hypertree decomposition seems less natural than that of query decomposition, this more powerful notion is the decomposition method for hypertrees that is more analogous to tree decomposition for graphs in terms of function. That is, hypergraphs of bounded hypertree-width are recognizable in polynomial-time, and therefore, a bounded width hypertree decomposition for a hypergraph of bounded hypertree-width can be constructed in polynomial-time. Hypertree decompositions differ from query decompositions only in these final two rules. Rule four in the definition of hypertree decomposition is a relaxation of rule four in the definition of query decomposition that allows for smaller widths in general (for example, in formula 4.5). Rule five in the definition of hypertree decomposition is a constraint that is needed for polynomial-time recognizability. In fact, there is a notion of *generalized hypertree decomposition* that is the same as the notion of hypertree decomposition but without rule five. However, like query decomposition, it is **NP**-complete to decide whether or not a hypergraph has generalized hypertree-width 4 (Gottlob, personal communication). So it seems that rule five is needed for tractability.

Note that formula 4.5 has hypertree-width 2 because it has the minimal hypertree decomposition of width 2 shown in the image above. Notice how atom J is "reused" with a subset of its variables in order to achieve the hypertree-width of 2 over the query-width

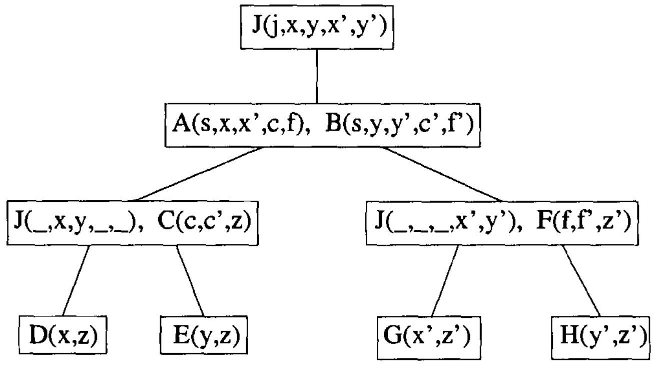

Figure 4.6: hypertree decomposition of formula 4.5

3 by this relaxation provided by rule four. The blanks in each node n of $\mathcal{T}$ are the elements of $vertices(\lambda(n)) \setminus \chi(n)$. Note that it is this hypertree decomposition that implies that formula 4.5 can be replaced by the semantically equivalent formula of query-width 2. This semantically equivalent formula is obtained by pushing projections down to the appropriate level of the formula as dictated by the hypertree decomposition, and then putting "anonymous variables" where the blanks are, that is, new variables that appear nowhere in formula 4.5 (*i.e.*, in Figure 4.6). In general, this is the case, that any formula ϕ that has hypertree-width k can be replaced by a semantically equivalent ϕ' such that model checking for ϕ' can be done in time $O(\ell n^k)$ where n is the size of the structure. Conversely, any formula that has the structure of a hypertree decomposition of width k (*i.e.*, all existentials are pushed inwards in a way that no join will involve more than k atoms) obviously has hypertree-width k. In fact, there is a fragment $\exists\text{FO}_{\wedge,+}$ (the fragment of FO logic that has only conjunction, and existential quantification) that pertains to exactly all of the formulas of $\exists\text{FO}_{\wedge,+}$ of bounded hypertree-width, namely the *k-guarded fragment* $\text{GF}_k(\exists\text{FO}_{\wedge,+})$ [2].

Definition 4.5 (the k-guarded fragment of $\exists\text{FO}_{\wedge,+}$). The k-guarded fragment of $\exists FO_{\wedge,+}$, *or* $GF_k(\exists FO_{\wedge,+})$, *is the smallest set of* $\exists FO_{\wedge,+}$ *formulas such that*

1. $GF_k(\exists FO_{\wedge,+})$ *contains atomic formulas;*

2. $GF_k(\exists FO_{\wedge,+})$ *is closed under Boolean operations;*

3. $GF_k(\exists FO_{\wedge,+})$ *contains* $\exists\bar{x}(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, *provided that the* G_i *are atomic formulas,* $m \leq k$, $\phi \in GF_k(\exists FO_{\wedge,+})$, *and the free variables of* ϕ *appear in the* G_i. *Here* $G_1 \wedge \ldots \wedge G_m$ *is called the* guard *of* ϕ.

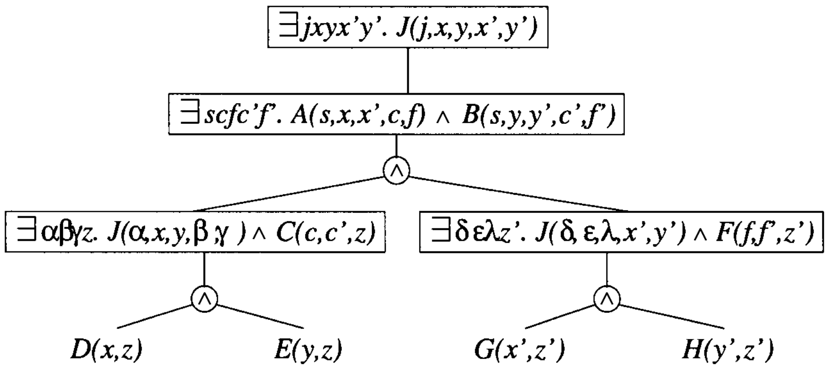
∃jxyx'y'. J(j,x,y,x',y')
∃scfc'f'. A(s,x,x',c,f) ∧ B(s,y,y',c',f')
∧
∃αβγz. J(α,x,y,β,γ) ∧ C(c,c',z)
∃δελz'. J(δ,ε,λ,x',y') ∧ F(f,f',z')
∧
D(x,z)
E(y,z)
∧
G(x',z')
H(y',z')

Figure 4.7: 2-guarded formula semantically equivalent to formula 4.5

This is simply the recursive definition for a tree-like formula, where no more than k atoms (or hyperedges) are associated with any vertex of this tree. If we view the formula as a tree, it is all of the internal vertices of the tree that are guards. Figure 4.7 shows the 2-guarded formula that is semantically equivalent to formula 4.5 (the one whose query decomposition is shown in Figure 4.5) written as a tree, where rectangular internal nodes stand for guarded existentials, circular internal nodes for conjunctions, and leaves for literals. So to restate, if formula ϕ has hypertree-width k, it means that it can be replaced by a semantically equivalent k-guarded ϕ'. Conversely, if ϕ is a k-guarded formula, it has hypertree-width k.

Note that we have only mentioned $\exists\text{FO}_{\wedge,+}$ throughout this entire chapter. However, in [24] the authors lift the expressibility of the k-guarded fragment to full FO logic by a proof technique involving stratified datalog that is inspired by [18]. That is, they show that model checking for formulas of the k-guarded fragment of FO logic, GF_k, can be done in polynomial-time.

Definition 4.6 (the k-guarded fragment). The k-guarded fragment, *or GF_k, is the smallest set of FO formulas such that*

1. *GF_k contains atomic formulas;*
2. *GF_k is closed under Boolean operations;*
3. *GF_k contains $\exists\bar{x}(G_1 \wedge \ldots \wedge G_m \wedge \phi)$, provided that the G_i are atomic formulas, $m \leq k$, $\phi \in GF_k$, and the free variables of ϕ appear in the G_i. Here $G_1 \wedge \ldots \wedge G_m$ is called the* guard *of ϕ.*

Note here that these formulas may contain negation, and disjunction (and thus universal quantification, etc.), however the guards must still be positive conjunctive.

In [46], the authors propose a recursive algorithm for evaluating formulas in GF_k w.r.t. structures (model checking) that has time $O(\ell n^k)$. Specifically, this algorithm takes formulas of the *strictly k-guarded fragment* SGF_k.

Definition 4.7 (the strictly k-guarded fragment). The strictly k-guarded fragment, *or SGF_k, are the formulas of GF_k of the form $\exists\bar{x}(G_1 \wedge \cdots \wedge G_m \wedge \psi)$.*

The degenerate cases where $\bar{x}$ is empty, $m = 0$ or ϕ is *true* fall in to this class as well. Note that any k-guarded sentence is strictly k-guarded. The algorithm is as follows. Note that there are two Eval procedures, but since one has two parameters and the other has three, there is never an ambiguitiy in which one is being used

Procedure Eval$(\mathcal{A}, \phi)$
Input: A structure $\mathcal{A}$ and a formula $\phi \in \mathrm{SGF}_k$
Output: $\phi(\mathcal{A})$

suppose $\phi(\bar{x}) = \exists\bar{y}(G_1 \wedge \cdots \wedge G_m \wedge \psi)$
return $\pi_{\bar{x}}\{\mathrm{Eval}(\mathcal{A}, G_1(\mathcal{A}) \bowtie \cdots \bowtie G_m(\mathcal{A}), \psi')\}$ where ψ' is the result of pushing $\neg$s in ψ inwards so that they are in front of atoms, equalities, or existentials.

Procedure Eval$(\mathcal{A}, \mathcal{R}, \phi)$ is defined recursively by:

1. If ϕ is an atom or an equality, then
   $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie \phi(\mathcal{A})$;
2. If ϕ is negation of an atom or an equality, then
   $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie^c \phi(\mathcal{A})$;
3. $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, (\phi \wedge \psi)) = \mathrm{Eval}(\mathcal{A}, \mathcal{R}, \phi) \cap \mathrm{Eval}(\mathcal{A}, \mathcal{R}, \psi)$;
4. $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, (\phi \vee \psi)) = \mathrm{Eval}(\mathcal{A}, \mathcal{R}, \phi) \cup \mathrm{Eval}(\mathcal{A}, \mathcal{R}, \psi)$;
5. $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, \exists\bar{y}\phi) = \mathcal{R} \bowtie \mathrm{Eval}(\mathcal{A}, \exists\bar{y}\phi)$;
6. $\mathrm{Eval}(\mathcal{A}, \mathcal{R}, \neg\exists\bar{y}\phi) = \mathcal{R} \bowtie^c \mathrm{Eval}(\mathcal{A}, \exists\bar{y}\phi)$.

where join ($\bowtie$) and Y-projection (π_Y) have the same meaning, but there are three new operators, *join with complement* ($\bowtie^c$), *intersection* ($\cap$) and *union* ($\cup$). These three new ones are here because of negation and disjunction. Note here that they present an algorithm for evaluating strictly k-guarded formulas instead of just k-guarded formulas. The reason they considered this is because their algorithm is recursively defined. In fact, the polynomial time combined complexity results ($O(\ell^2 n^k)$ by Gottlob *et al.*, and this one only applies to strictly k-guarded formulas not k-guarded formulas in general. To see why, by the definition of GF_k, $\neg R(x, y, z)$ is a 1-guarded formula, but it cannot be evaluated in $O(n)$ time. However, since most problems in practice are sentences, restricting formulas to the strictly k-guarded fragment is not a drawback.

The join with complement $\mathcal{R} \bowtie^c \mathcal{S}$ of X-relation $\mathcal{R}$ and Y-relation $\mathcal{S}$ is the $X \cup Y$-relation $\{\gamma : X \cup Y \to A \mid \gamma|_X \in \mathcal{R}, \gamma|_Y \notin \mathcal{S}\}$. This pertains to an atom conjoined with the negation of another atom, more precisely, if $\mathcal{R}$ is $\phi_1(\mathcal{A})$ and $\mathcal{S}$ is $\phi_2(\mathcal{A})$, then $\mathcal{R} \bowtie^c \mathcal{S}$ is $(\phi_1 \wedge \neg\phi_2)(\mathcal{A})$. Join with complement is an $O(\|\mathcal{R}\| \cdot \|\mathcal{S}\|)$-time operator, but when $Y \subseteq X$, it is an $O(\|\mathcal{R}\| + \|\mathcal{S}\|)$-time operator for a similar reason as with join. When a join needs to be performed between two X-relations, it is the same as a list intersection (which is obviously linear time as it falls under the special case of join, *i.e.*, $X \subseteq X$). It is this case that the intersection operator refers to. Finally union handles disjunction. The union $\mathcal{R} \cup \mathcal{S}$ of X-relation $\mathcal{R}$ and X-relation $\mathcal{S}$ is the X-relation $\{\gamma : X \to A \mid \gamma \in \mathcal{R} \text{ or } \gamma \in \mathcal{S}\}$. This is also just the merging of two sorted lists, and thus is an $O(\|\mathcal{R}\| + \|\mathcal{S}\|)$-time operator. From the definition of SGF_k and the form of the procedure Eval it can be seen that all join operations fall under the special case of join, that all negated atoms are "guarded", that is all joins with complement fall under the special case as well, and that all disjunctions are between X-relations and X-relations (and can therefore be handled by the union operator). In other words, Eval returns the correct answer $\phi(\mathcal{A})$, and furthermore, since all operators are linear time, model checking for a strictly k-guarded formula can be done in time $O(\ell n^k)$ by using the same argument as with query decomposition. Namely, evaluating each guard (where guards correspond exactly to the vertices $\mathcal{T}$ of the hypertree decomposition), involves doing no more than k joins. Now we have to do a series of $O(\ell)$ linear time operations on relations of size $O(n^k)$, which means that all intermediate relations will be of size $O(n^k)$. Thus the overall time is $O(\ell n^k)$. It is this model checking algorithm for GF_k that we use as a basis for our grounding algorithm for model expansion.

On a related note, earlier in the chapter we mentioned that $\exists\mathrm{FO}_{\wedge,+}$ formulas of *tree-width*

k can be evaluated with polynomial-time combined complexity (w.r.t. model checking), that is, in time $O(\ell n^k)$. In fact, it has been shown in [36] that the set of formulas of bounded tree-width have a corresponding logic, namely the set of $\exists\mathrm{FO}_{\wedge,+}$ formulas that use no more than k-variables. It is the expressibility of the k-variable fragment that the authors of [18] lift to full FO logic using the proof technique involving stratified datalog. That is, they show that model checking for k-variable FO logic, or FO^k, can be done in polynomial-time.

Chapter 5

Grounding Mechanism for Model Expansion

In this chapter, we outline our approach to solving model expansion problems. Essentially we obtain a *propositionalization* (or *grounding*) of the FO formula over the structure that has a satisfying assignment for every expansion structure that satisfies the FO formula. We then use a propositional satisfiability solver to find satisfying assignments for the grounding (if any exist), which determines which expansion structures satisfy the FO formula (if any). For this, in Section 5.1, we present this approach in comparison to the related approaches in general. In Section 5.2, we give the representation that we use for the grounding of a formula over a structure. Finally, in Section 5.3, we provide an algebra for these representations: a way to combine representations for each connective of FO logic. As such, with this mechanism, we can now develop recursive algorithms for computing the grounding of a formula. This is done by starting at the representations of the atomic subformulas of a formula, and working inductively on the structure of the formula, combining representations according to the connectives of the formula in order to arrive at such a representation for the entire formula, *i.e.*, a grounding of the formula over the structure.

5.1 Grounding for Model Expansion

An approach to solving a problem specified in a high-level language is to "compile" it into a lower level language in a way that there is a one to one correspondence between solutions to the problem specified in the high-level language to solutions to the compilation in the lower level language. Then we use an optimized efficient low-level solver to obtain solutions to the lower level compilation, *i.e.* solutions to the original problem. In the context of model expansion, given a FO formula and a structure, this is done by *propositionalizing* (or *grounding*) the combination of problem instance and description in such a way that the satisfying assignments of this grounding correspond exactly to the expansion structures that satisfy the FO formula. We then use one of the many efficient solvers for propositional satisfiability for obtaining solutions. Given the efficiency of modern SAT technology [51], all that is needed is a an efficient grounding algorithm; this is a practical approach to solving model expansion problems. This is the main topic of this thesis, our main contribution is an efficient grounding algorithm for the model expansion framework.

Grounding for any high-level language is the process of eliminating variables (and quantifiers) by replacing them with constant symbols. The approach of grounding is taken in ASP and logic programming in general, however this approach differs from ours. The approach of ASP and logic programming uses *Herbrand models*, which is done by taking the *Herbrand universe* and then creating ground instances over this universe. The Herbrand universe of a logic program has an element for every term of the logic program. So if a is some term of the program, and f is some function, then the Herbrand universe contains $a, f(a), f(f(a)), f(f(f(a))), \ldots$, so in general, this universe can be infinite, causing satisfiability to become undecidable. Thus, for ASP and logic programming, restrictions must be put in place when function symbols are used, in order to guarantee decidability.

Our grounding approach, however, relies on the fact that in the MX framework, there is a separation of instance and problem description. That is, the instance, a finite structure, provides the domain onto which every term falls. That is, if a is some term of the formula, and f is some function, then $a, f(a), f(f(a)), f(f(f(a))), \ldots$, all fall onto the domain A of the structure $\mathcal{A} = (A; \sigma^{\mathcal{A}})$. So in our approach, the universe is the domain of the structure, and thus, there need not be any restrictions on functions. That is, instead of using Herbrand models, in grounding, we bring domain elements into the syntax by expanding the vocabulary and associating a new constant symbol with each element of the domain. For domain A, we denote the set of such constants as $\tilde{A}$. We define a *grounding* as

Definition 5.1.1 (grounding for MX). *Formula ψ is a* grounding *of formula ϕ over σ-structure $\mathcal{A} = (A; \sigma^{\mathcal{A}})$ if*

1. *ψ is a ground formula, i.e., there are no FO variables in ψ*
2. *ψ is over vocab(ϕ) $\cup \tilde{A}$; and*
3. *for every expansion structure $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ over vocab(ϕ), $\mathcal{B} \models \phi$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi$, where $(\mathcal{B}, \tilde{A}^{\mathcal{B}})$ is the structure obtained by expanding $\mathcal{B}$ by interpretations of the new constants $\tilde{A}$.*

While obtaining the above is the intent of our approach, there are many ways of obtaining a grounding of this type as well. The most naive algorithm for grounding a FO sentence ϕ over a structure $\mathcal{A}$ is to first re-write ϕ in prenex normal form, $\phi = Q_1 x_1 \cdots Q_m x_m \theta$, then to recursively construct $\psi = G(\phi)$ as follows

$$G(\phi) = \begin{cases} \phi & \text{if } \phi \text{ is } Q\text{-free,} \\ \bigvee_{a \in A} G(\phi') & \text{if } \phi \text{ is } \exists x \phi' \\ \bigwedge_{a \in A} G(\phi') & \text{if } \phi \text{ is } \forall x \phi' \end{cases}$$

The time required by this algorithm is $O(n^{|\phi|})$, where n is the size of structure $\mathcal{A}$. While this time is exponential in the length of the formula, it is also the upper bound on the size of the grounding itself. While there are ways to improve the running time and grounding size, such as restricting the form of the formula, the bound of this naive algorithm will be the asymptotic bound of the grounding for any approach for grounding a FO formula over a finite FO structure.

To make this approach practical, we want to simplify the grounding as much as possible, before we pass it to a SAT solver for solution. While the definition above states that a grounding ψ is over *vocab*(ϕ) $\cup \tilde{A}$, there is no need for any grounding to contain symbols of σ, as the truth value for ground atoms of σ can be determined by the structure (without having to resort to the SAT solver). It is thus favorable to "evaluate out" these "known" parts of the formula, leaving a ground formula in the expansion vocabulary only, which we call a *reduced grounding.*

Definition 5.1.2 (reduced grounding for MX). *Let ψ be a grounding of ϕ over σ-structure $\mathcal{A}$. Then ψ is a* reduced grounding *if it only contains symbols of $\tilde{A}$ and of the expansion vocabulary ε (i.e., no symbols of σ).*

Given a grounding as defined above, this can be transformed into a reduced grounding using a straight-forward, but brute-force method. For each k-ary instance relation R and each k-tuple $\bar{t}$ over $\tilde{A}$, if $\bar{t} \in R^{\mathcal{A}}$, (resp. $\bar{t} \notin R^{\mathcal{A}}$), replace each occurrence of $R(\bar{t})$ in ϕ with *true* (resp. *false*). Recursively eliminate occurrences of *true* and *false* by replacing $(\psi \wedge true)$ with (ψ), etc. In implementing our approach of combining grounding with partial evaluation, however, it would be best to remove the contributions of σ during the construction of ψ. While doing so does not change the asymptotic time or size bound in general, it is a good practice. The mechanism we show for grounding FO formulas in the rest of this chapter has partial evaluation built into it, so any algorithm using this mechanism will do this automatically, to produce a reduced grounding.

5.2 Extended Relations

In this section we describe how to represent the grounding of a FO formula over a finite FO structure. In order to have a recursive algorithm for grounding a formula ϕ over structure $\mathcal{A}$ (computing the representation for the entire formula), we first compute representations for all atomic subformulas of ϕ (w.r.t. $\mathcal{A}$), *i.e.*, we can represent formulas with free variables. Then we need a way of combining representations according to the different connectives of FO logic, or an *algebra*, which is explained in the next section. Once we have all of this, we then have a mechanism for grounding formulas over structures. An algorithm for grounding is then a recipe that uses this machinery to generate a representation for the entire formula. In the next chapter, we present such an algorithm; specifically, our algorithm is defined in such a way that if the formula has a certain form, the algorithm, using this mechanism, is guaranteed to produce a representation (grounding) within a certain time bound.

We use *extended relations* for representing the grounding of a formula over a structure, which is an extension of the notion of relation from database theory. Recall from chapter 4 that we used the notion of a relation for representing solutions to model checking problems. In the examples of model checking problems, given a formula ϕ and a structure $\mathcal{A}$, testing if $\mathcal{A} \models \phi$, we computed relations for each atomic subformula of ϕ. We then combined these representations according to the algebra for relations, the set of operations corresponding to each connective of the formula. In this vein, our approach is very similar, only the difference is that since in our case there are expansion predicates in the formula, instead of producing a set of solutions at each step, we are producing a set of groundings at each step. More specifically, for model checking, a relation for a formula ϕ w.r.t. structure $\mathcal{A}$ is a set mappings from the free variables of ϕ to the domain elements A (of structure $\mathcal{A}$). What this is, is the set of valuations of the free variables such that the formula is satisfiable in the structure. On the other hand, for model expansion, the extended relation is defined as

Definition 5.2.1 (extended X-relation). *Let $\mathcal{A} = (A; \sigma^{\mathcal{A}})$, and X be the set of free variables of formula ϕ. An extended X-relation $\mathcal{R}$ over A is a set of pairs (γ, ψ) s.t.*

1. *ψ is a ground formula over $\varepsilon \cup \tilde{A}$ and $\gamma : X \rightarrow A$;*
2. *for every γ, there is at most one ψ s.t. $(\gamma, \psi) \in \mathcal{R}$.*

As with an X-relation, an extended X-relation $\mathcal{R}$ has a set of mappings γ from the set of variables X to the set of domain elements A. But furthermore, extending the notion of X-relation, extended X-relation $\mathcal{R}$ is a unique mapping from all possible instantiations of variables in X to ground formulas ψ. So, for model expansion, an extended relation for a formula ϕ w.r.t. structure $\mathcal{A}$ is a set of mappings from the free variables of ϕ to the domain elements of A, and for each of these mappings, there is an associated ground formula ψ. Note that for mappings γ not appearing in $\mathcal{R}$, the associated formula is $false$. For extended X-relation $\mathcal{R}$, we will simply write $\gamma \in \mathcal{R}$ to mean that there exists a ψ such that $(\gamma, \psi) \in \mathcal{R}$. Formally, the mapping represented by $\mathcal{R}$ is

Definition 5.2.2 (mapping represented by $\mathcal{R}$). *Let $\mathcal{R}$ be an extended X-relation over A. The unique mapping represented by $\mathcal{R}$, denoted by $\delta_{\mathcal{R}}$, is defined as follows. Let $\gamma : X \rightarrow A$. Then*

$$\delta_{\mathcal{R}}(\gamma) = \begin{cases} \psi & \text{if } (\gamma, \psi) \in \mathcal{R} \\ false & \text{if } \gamma \notin \mathcal{R} \end{cases}$$

The following is an example of an extended $\{x, y, z\}$-relation $\mathcal{R}$, where we will take $\{0, \ldots, 9\}$ as our domain A (where E would be a member of the expansion vocabulary ε)

x	y	z	ψ
1	2	3	$E(1,2) \wedge E(2,3)$
3	4	5	$E(3,4) \wedge E(4,5)$

Here, $\mathcal{R}$ is a unique mapping from all possible instantiations of variables in $\{x, y, z\}$ to ground formulas in that it maps (1,2,3) to $E(1,2) \wedge E(2,3)$, that is, if $\gamma = (1,2,3)$ (we use a tuple to represent a particular γ) then $\delta(\gamma) = E(1,2) \wedge E(2,3)$. It maps (3,4,5) to

$E(3,4) \wedge E(4,5)$, and otherwise, $\delta(\gamma) = false$. As a convention, we refer to the columns of the table concerning the variables as the "relation part" of the extended relation, and the column concerning the formulas as the "formula part".

Note that the γ other than (1,2,3) or (3,4,5) are not even represented in $\mathcal{R}$. This is due to the fact that we write $\gamma \in \mathcal{R}$ to mean that there exists ψ such that $(\gamma, \psi) \in \mathcal{R}$. Since we created this example, the above table is the most compact way to represent $\mathcal{R}$ (we represent these sets of mappings with tables), however in general, our definition of an extended relation $\mathcal{R}$ allows the possibility that for some γ appearing in $\mathcal{R}$, the associated formula is the propositional symbol $false$ or is equivalent to $false$. For example, the following is an example of an extended $\{x, y, z\}$-relation $\mathcal{S}$, where $A = \{0, \dots, 9\}$

x	y	z	ψ
1	2	3	$E(1,2) \wedge E(2,3)$
2	2	4	$false$
3	4	5	$E(3,4) \wedge E(4,5)$
8	9	1	$P \wedge \neg P$

however, it is easy to realize that $\mathcal{S}$ is equivalent to $\mathcal{R}$ according to the definition of extended relation.

Also, for extended relations, note that for some γ appearing in $\mathcal{R}$, the associated formula is the propositional symbol $true$. In fact, an ordinary relation can be represented as an extended relation where the formula associated with each mapping γ in the relation is $true$, (where the rest are $false$). Also, a single ground formula ψ can be represented as an extended relation where the set of variables is empty and the formula associated with this single empty mapping is ψ.

As mentioned earlier, given a formula ϕ in $\sigma \cup \varepsilon$ with free variables X and a σ-structure $\mathcal{A}$, we will use an extended X-relation to represent a reduced grounding of ϕ over $\mathcal{A}$ under each possible instantiation of variables. We will call such an extended relation *an answer to ϕ w.r.t. $\mathcal{A}$*, which we now formally define. Let $\gamma : X \rightarrow A$. We will use $\phi[\gamma]$ to denote the result of instantiating free variables in ϕ according to γ.

Definition 5.2.3 (answer to ϕ w.r.t. $\mathcal{A}$). *Let ϕ be a formula in $\sigma \cup \varepsilon$ with free variables X, $\mathcal{A}$ a σ-structure with domain A, and $\mathcal{R}$ an extended X-relation over A. We say that $\mathcal{R}$ is* an answer to ϕ w.r.t. $\mathcal{A}$ *if for any $\gamma : X \rightarrow A$, we have that $\delta_{\mathcal{R}}(\gamma)$ is a reduced grounding of $\phi[\gamma]$ over $\mathcal{A}$.*

Note that this is analogous to the notion of answer in the ordinary relational context, where we are given a formula ϕ in σ with free variables X, $\mathcal{A}$ a σ-structure with domain A, and $\mathcal{R}$ an X-relation over A. We say that $\mathcal{R}$ is an answer to ϕ w.r.t. $\mathcal{A}$ if for any $\gamma: X \to A$, $\phi[\gamma] = true$.

As a special case of the above definition for extended relations, an answer to a sentence ϕ consists of a single formula, which is a reduced grounding of ϕ. While we can represent answers to formulas in the general case, we focus on this special case because we are interested in doing grounding for FO sentences, and passing this grounding (the single formula) to a propositional satisfiability solver for solution. We now give an example of the production of a reduced grounding of a sentence to better illustrate the notion of extended relation and how it represents the answer to a formula w.r.t. a structure.

Let $\phi = \exists x \exists y \exists z[P(x,y,z) \wedge E(x,y) \wedge E(y,z)]$, $\sigma = \{P\}$, and $\varepsilon = \{E\}$. Let $\mathcal{A}$ be a σ-structure such that $P^{\mathcal{A}} = \{(1,2,3),(3,4,5)\}$. Then the extended relation $\mathcal{R}$ shown in the table above is an answer to $\phi' = P(x,y,z) \wedge E(x,y) \wedge E(y,z)$ w.r.t. $\mathcal{A}$. It is easy to see, for example, that $E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi'[(1,2,3)] = P(1,2,3) \wedge E(1,2) \wedge E(2,3) = true \wedge E(1,2) \wedge E(2,3) = E(1,2) \wedge E(2,3)$, and $false$ is a reduced grounding of $\phi'[(1,1,1)]$. Next, the following extended relation is an answer to $\phi'' = \exists z[P(x,y,z) \wedge E(x,y) \wedge E(y,z)]$

x	y	ψ
1	2	$E(1,2) \wedge E(2,3)$
3	4	$E(3,4) \wedge E(4,5)$

Here, for example, $E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi''[(1,2)] = \exists z[P(1,2,z) \wedge E(1,2) \wedge E(2,z)] = P(1,2,3) \wedge E(1,2) \wedge E(2,3) = true \wedge E(1,2) \wedge E(2,3) = E(1,2) \wedge E(2,3)$, and $false$ is a reduced grounding of $\phi''[(1,1)]$. Finally, the following is an answer to ϕ, where the single formula is a reduced grounding of ϕ

ψ
$[E(1,2) \wedge E(2,3)] \vee [E(3,4) \wedge E(4,5)]$

Now that we have defined what we use to represent an answer to a formula w.r.t. a structure, namely an extended relation, we need simply to define the algebra for extended relations in order to have a mechanism for grounding an FO formula over a structure. We define this algebra and explain it in full in the next section.

5.3 An Algebra for Extended Relations

In the previous section, we have defined the extended relation, which extends the notion of relation from database theory. Just as relations have an algebra, *i.e.*, a set of operations whose semantics correspond to the connectives of FO logic defined on them, we define an algebra for extended relations, since we want to develop algorithms for model expansion (grounding), similar to the existing algorithms for model checking.

In traditional relational algebra, there are the operations join, join with complement, projection, union and intersection, which correspond to conjunction, conjunction with a negated atom, existential quantification, disjunction, and a special case of conjunction respectively. Detailed algorithms and proofs of the operations join and projection can be found in the appendix of [18], and for the rest of the operations it follows quite easily. The essential idea here is to encode the domain elements appearing in the input relation using the bucket sort algorithm (which is possible, since the domain is fixed for a given instance of the problem). Next, we compute a join, projection, union, etc. of the encoded input relations. Finally we sort (if necessary) the encoded output relation, and then decode this result. It can be shown that each of the operations in the ordinary relational algebra can be done in time linear in its operands, with the exception of join and join with complement. However, in the case where the first operand is an X-relation and the second operand is a Y-relation, join and join with complement can be done in time linear in these operands if $Y \subseteq X$.

In this section, we develop an algebra for extended relations which consists of these five operations, where each of these has the same semantics as in traditional relational algebra. In the following, we present each operation, state its semantics, and give the time complexity. We will show that each of these operations can be done in time linear in its operands, excluding join and join with complement. We show that join and join with complement, if the operands meet the same criteria for extended relations as above for ordinary relations, the time of these operations is quadratic it their operands. However, after presenting the details of the algorithm in the next chapter, we show how to modify the algorithm in order to avoid any size blow-up from these operations. The major difference between the algebra we present, and the ordinary relational algebra, is since we are dealing with extended relations (those where formulas are associated with each mapping), we must define our operations in terms of these formulas, and thus copy formulas from the input relations to the output relation.

While a notion of the size of a relation [18] is needed in order to prove bounds for the complexity of any operation in the relational algebra, we also give a measure of the size of an extended relation. Let $\mathcal{R}$ be an extended X-relation. The size of $\mathcal{R}$, denoted by $||\mathcal{R}||$, is the sum of the size of each tuple $(\gamma, \delta_{\mathcal{R}}(\gamma))$ in $\mathcal{R}$. Each tuple is of size $|X| + ||\delta_{\mathcal{R}}(\gamma)||$, where $||\delta_{\mathcal{R}}(\gamma)||$ is the size of the formula associated with γ. Thus $||\mathcal{R}|| = |X||\mathcal{R}| + \sum_{\gamma \in \mathcal{R}} ||\delta_{\mathcal{R}}(\gamma)||$, where $|\mathcal{R}|$ is the number of tuples in $\mathcal{R}$. We now define the five operations of the algebra for extended relations starting with join and join with complement.

Definition 5.3.1 (join, join with complement). *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A. Then*

1. *the* join *of $\mathcal{R}$ and $\mathcal{S}$ is the extended $X \cup Y$-relation

$$\mathcal{R} \bowtie \mathcal{S} = \{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in \mathcal{R}, \gamma|_Y \in \mathcal{S}, \text{ and } \psi = \delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y)\};$$

2. *the* join *of $\mathcal{R}$* with the complement *of $\mathcal{S}$ is the extended $X \cup Y$-relation*

$$\mathcal{R} \bowtie^c \mathcal{S} = \{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in \mathcal{R}, \text{ and } \psi = \delta_{\mathcal{R}}(\gamma|_X) \wedge \neg\delta_{\mathcal{S}}(\gamma|_Y)\}.$$

As an example of the operations join and join with complement, let $\mathcal{R}$ be the extended $\{x, y, z\}$-relation

x	y	z	ϕ
3	2	1	$A \wedge B$
4	2	1	$E \vee F$
4	6	8	E
7	7	9	$C \vee D$

and $\mathcal{S}$ be the extended $\{y, z\}$-relation ($\{y, z\}$ is a subset of $\{x, y, z\}$)

y	z	ϕ
2	1	$\neg C$
3	4	$F \wedge B$
7	9	$\neg A \wedge \neg E$

The join of $\mathcal{R}$ and $\mathcal{S}$, $\mathcal{R} \bowtie \mathcal{S}$, is the extended $\{x, y, z\}$-relation

x	y	z	ϕ
3	2	1	$(A \wedge B) \wedge \neg C$
4	2	1	$(E \vee F) \wedge \neg C$
7	7	9	$(C \vee \neg D) \wedge (\neg A \wedge \neg E)$

The join of $\mathcal{R}$ with the complement of $\mathcal{S}$, $\mathcal{R} \bowtie^c \mathcal{S}$, is the extended $\{x, y, z\}$-relation

x	y	z	ϕ
3	2	1	$(A \wedge B) \wedge C$
4	2	1	$(E \vee F) \wedge C$
4	6	8	E
7	7	9	$(C \vee \neg D) \wedge \neg(\neg A \wedge \neg E)$

The following proposition states the semantics of these operations.

Proposition 5.3.2. *Suppose that $\mathcal{R}$ is an answer to ϕ_1 and $\mathcal{S}$ is an answer to ϕ_2, both w.r.t. structure $\mathcal{A}$. Then*

1. *$\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ w.r.t. $\mathcal{A}$;*

2. *$\mathcal{R} \bowtie^c \mathcal{S}$ is an answer to $\phi_1 \wedge \neg\phi_2$ w.r.t. $\mathcal{A}$.*

Proof: (1) Since $\mathcal{R}$ is an answer to ϕ_1 w.r.t. $\mathcal{A}$, then for any instantiation γ_1 of the variables X, $\delta_{\mathcal{R}}(\gamma_1)$ is a reduced grounding of $\phi_1[\gamma_1]$. Since $\mathcal{S}$ is an answer to ϕ_2 w.r.t. $\mathcal{A}$, then for any instantiation γ_2 of the variables Y, $\delta_{\mathcal{S}}(\gamma_2)$ is a reduced grounding of $\phi_2[\gamma_2]$. Firstly, $\delta_{\mathcal{R}}(\gamma_1) \wedge \delta_{\mathcal{R}}(\gamma_2)$ is a ground formula over $\varepsilon \cup \tilde{A}$, since both $\delta_{\mathcal{R}}(\gamma_1)$ and $\delta_{\mathcal{R}}(\gamma_2)$ are ground formulas over $\varepsilon \cup \tilde{A}$. Secondly, since $\delta_{\mathcal{R}}(\gamma_1)$ is such that for every expansion structure $\mathcal{B}_1 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_1})$ over *vocab*(ϕ_1) (or even a superset of this vocabulary), $\mathcal{B}_1 \models \phi_1[\gamma_1] \Leftrightarrow (\mathcal{B}_1, \tilde{A}^{\mathcal{B}_1}) \models \delta_{\mathcal{R}}(\gamma_1)$, and $\delta_{\mathcal{S}}(\gamma_2)$ is such that for every expansion structure $\mathcal{B}_2 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_2})$ over *vocab*(ϕ_2), $\mathcal{B}_2 \models \phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_2, \tilde{A}^{\mathcal{B}_2}) \models \delta_{\mathcal{S}}(\gamma_2)$, it follows that for every expansion structure $\mathcal{B}_3 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_3})$ over *vocab*$(\phi_1) \cup$ *vocab*(ϕ_2), $\mathcal{B}_3 \models \phi_1[\gamma_1]$ and $\mathcal{B}_3 \models \phi_2[\gamma_2]$ iff $(\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{R}}(\gamma_1)$ and $(\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{S}}(\gamma_2)$. By the semantics of "$\wedge$", it follows that for every expansion $\mathcal{B}_3$, $\mathcal{B}_3 \models \phi_1[\gamma_1] \wedge \phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{R}}(\gamma_1) \wedge \delta_{\mathcal{S}}(\gamma_2)$, and therefore $\delta_{\mathcal{R}}(\gamma_1) \wedge \delta_{\mathcal{S}}(\gamma_2)$ is a reduced grounding of $(\phi_1 \wedge \phi_2)[\gamma_3]$ where $\gamma_3|_X = \gamma_1$, $\gamma_3|_Y = \gamma_2$. Thus $\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ w.r.t. $\mathcal{A}$. (2) Since for every expansion $\mathcal{B}_2$ over *vocab*(ϕ_2), $\mathcal{B}_2 \models \phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_2, \tilde{A}^{\mathcal{B}_2}) \models \delta_{\mathcal{S}}(\gamma_2)$, for every expansion $\mathcal{B}'_2$ over *vocab*(ϕ_2), $\mathcal{B}'_2 \not\models \phi_2[\gamma_2] \Leftrightarrow$

$(\mathcal{B}_2', \tilde{A}^{\mathcal{B}_2'}) \not\models \delta_\mathcal{S}(\gamma_2)$. Thus, by the semantics of "$\neg$", for every expansion $\mathcal{B}_2'$ over *vocab*(ϕ_2), $\mathcal{B}_2' \models \neg\phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_2', \tilde{A}^{\mathcal{B}_2'}) \models \neg\delta_\mathcal{S}(\gamma_2)$. Therefore, by the semantics of "$\wedge$", it follows that for every expansion $\mathcal{B}_3'$ over *vocab*$(\phi_1) \cup$ *vocab*(ϕ_2), $\mathcal{B}_3' \models \phi_1[\gamma_1] \wedge \neg\phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_3', \tilde{A}^{\mathcal{B}_3'}) \models \delta_\mathcal{R}(\gamma_1) \wedge \neg\delta_\mathcal{S}(\gamma_2)$, and therefore $\delta\mathcal{R}(\gamma_1) \wedge \neg\delta_\mathcal{S}(\gamma_2)$ is a reduced grounding of $(\phi_1 \wedge \neg\phi_2)[\gamma_3]$ where $\gamma_3|_X = \gamma_1$. Thus $\mathcal{R} \bowtie^c \mathcal{S}$ is an answer to $\phi_1 \wedge \neg\phi_2$ w.r.t. $\mathcal{A}$. ■

The following proposition states the time complexity of any reasonable implementation of the operations join and join with complement in the case where $Y \subseteq X$. We only consider the complexity for this case because in the main algorithm we present in the next Chapter, this will always be the case for join and join with complement.

Proposition 5.3.3. *For extended X-relation $\mathcal{R}$ and extended Y-relation $\mathcal{S}$, when $Y \subseteq X$, join ($\mathcal{R} \bowtie \mathcal{S}$) and join with complement ($\mathcal{R} \bowtie^c \mathcal{S}$) both have time complexity* $O(\|\mathcal{R}\| \cdot \|\mathcal{S}\|)$

Proof: Since $\mathcal{R}$ and $\mathcal{S}$ are over a fixed finite domain A, they can be sorted according to some total order on A in time $O(\|\mathcal{R}\|)$ and $O(\|\mathcal{S}\|)$ respectively, using bucket sort. Now the join (join with complement) of the sorted $\mathcal{R}$ and $\mathcal{S}$ can be computed in a fashion similar to merging two sorted lists. Start at the beginning of $\mathcal{R}$ and $\mathcal{S}$. Which ever entry γ of $\mathcal{R}$ or $\mathcal{S}$ has the higher lexical value according to the total order, move down the opposite extended relational table, computing any possible matches of tuples in each table until an entry with a higher lexical value is reached. Then move down the other extended relational table in the same fashion. Repeat this procedure until both tables have been traversed. In general, this procedure will generate an extended $X \cup Y$-relation of size $O(|X||\mathcal{R}| + \sum_{\gamma\in\mathcal{R}} \|\delta_\mathcal{R}(\gamma)\| + |\mathcal{R}| \cdot \sum_{\gamma\in\mathcal{S}} \|\delta_\mathcal{S}(\gamma)\|) \leq O(\|\mathcal{R}\| \cdot \|\mathcal{S}\|)$. ■

While the complexity of join and join with complement is $O(\|\mathcal{R}\| \cdot \|\mathcal{S}\|)$ for input $\mathcal{R}$ and $\mathcal{S}$, in the next section, after we present the main algorithm, we will show that, by modifying this algorithm slightly, any size blow-up from the operations join and join with complement can always be avoided.

We now present the projection operation.

Definition 5.3.4 (Y-projection). *Let $\mathcal{R}$ be an extended X-relation and $Y \subseteq X$. The* Y-projection *of $\mathcal{R}$, denoted by $\pi_Y(\mathcal{R})$, is the extended Y-relation*

$$\{(\gamma', \psi) \mid \gamma' = \gamma|_Y \textit{ for some } \gamma \in \mathcal{R} \textit{ and } \psi = \bigvee_{\{\gamma\in\mathcal{R} \mid \gamma|_Y = \gamma'\}} \delta_\mathcal{R}(\gamma).$$

We take for our example of the projection operation, a part of the example of the previous section, namely, let $\mathcal{R}$ be the extended $\{x, y\}$-relation

x	y	ψ
1	2	$E(1,2) \wedge E(2,3)$
3	4	$E(3,4) \wedge E(4,5)$

The $\emptyset$-projection of $\mathcal{R}$, $\pi_\emptyset(\mathcal{R})$, is the extended $\emptyset$-relation

ψ
$[E(1,2) \wedge E(2,3)] \vee [E(3,4) \wedge E(4,5)]$

Given the following proposition of the semantics of the projection operation, it should now be clearer how the process of grounding this sentence $\phi = \exists x \exists y \exists z[P(x,y,z) \wedge E(x,y) \wedge E(y,z)]$ in the previous section proceeds. For this previous example, we started with answer to $\phi' = P(x,y,z) \wedge E(x,y) \wedge E(y,z)$ and projected it onto the set $\{x,y\}$ for the answer to $\phi'' = \exists z[P(x,y,z) \wedge E(x,y) \wedge E(y,z)]$. We then projected this answer onto the set $\emptyset$ (as in the above example) for the answer to ϕ.

Proposition 5.3.5. *Suppose that $\mathcal{R}$ is an answer to ϕ w.r.t. $\mathcal{A}$, and Y is the set of free variables of $\exists \bar{z}\phi$. Then $\pi_Y(\mathcal{R})$ is an answer to $\exists \bar{z}\phi$ w.r.t. $\mathcal{A}$.*

Proof: Let $\gamma_1, \ldots, \gamma_m \in \mathcal{R}$ be the set of mappings that are equivalent to some γ when restricted to the set Y, *i.e.* $\gamma_1|_Y = \cdots = \gamma_m|_Y = \gamma$. Since $\mathcal{R}$ is an answer to ϕ w.r.t. $\mathcal{A}$ for each $\gamma_i \in \gamma_1, \ldots, \gamma_m$, $\delta_{\mathcal{R}}(\gamma_i)$ is a reduced grounding of $\phi[\gamma_i]$, it follows that $\delta_{\mathcal{R}}(\gamma_1), \ldots, \delta_{\mathcal{R}}(\gamma_m)$ are over $\varepsilon \cup \tilde{A}$, which means that $\delta_{\pi_Y(\mathcal{R})}(\gamma) = \delta_{\mathcal{R}}(\gamma_1) \vee \cdots \vee \delta_{\mathcal{R}}(\gamma_m)$ is over $\varepsilon \cup \tilde{A}$. Secondly, by the fact that $\mathcal{R}$ is an answer to ϕ w.r.t. $\mathcal{A}$, it follows that for each γ_i, $\delta_{\mathcal{R}}(\gamma_i)$ is such that for every expansion structure $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ over *vocab*(ϕ), $\mathcal{B} \models \phi[\gamma_i] \Leftrightarrow (\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \delta_{\mathcal{R}}(\gamma_i)$. For every expansion structure $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ over *vocab*(ϕ), $\mathcal{B} \models \exists \bar{z}\phi[\gamma] \Leftrightarrow$ for some valuation v of tuple $\bar{z}$ in domain A of $\mathcal{A}$, $\mathcal{B}[\bar{z} : v] \models \phi[\gamma]$, by the semantics of "$\exists$". $\mathcal{B}[\bar{z} : v] \models \phi[\gamma]$ for some v of $\bar{z} \Leftrightarrow \mathcal{B} \models \phi[\gamma_i]$ for some $\gamma_i \in \gamma_1 \ldots \gamma_m$ s.t. $\gamma_i(\bar{z}) = v$ (as these are the mappings that are equivalent to γ when restricted to Y), where we extend the mapping γ from variables to tuples of variables in the obvious way. $\mathcal{B} \models \phi[\gamma_i] \Leftrightarrow (\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \delta_{\mathcal{R}}(\gamma_i)$. Finally, by the semantics of "$\vee$", $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \delta_{\mathcal{R}}(\gamma_i)$ for some $\gamma_i \in \gamma_1, \ldots, \gamma_m \Leftrightarrow (\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \delta_{\mathcal{R}}(\gamma_1) \vee \cdots \vee \delta_{\mathcal{R}}(\gamma_m) = \delta_{\pi_Y(\mathcal{R})}(\gamma)$. Thus $\delta_{\pi_Y(\mathcal{R})}(\gamma)$ is a reduced grounding of $\exists \bar{z}\phi[\gamma]$ and therefore $\pi_Y(\mathcal{R})$ is an answer to $\exists \bar{z}\phi$ w.r.t. $\mathcal{A}$. ■

The following proposition states the time complexity of any reasonable implementation of the projection operation.

Proposition 5.3.6. *The projection operation has time complexity $O(\|\mathcal{R}\|)$.*

Proof: First make a pass through extended relational table $\mathcal{R}$ to eliminate the columns from the relation part of $\mathcal{R}$ that correspond to the variables in tuple $\bar{z}$, which takes time $O(\|\mathcal{R}\|)$. Since $\mathcal{R}$ is over a fixed finite domain A, we sort this intermediate table in time $O(\|\mathcal{R}\|)$ using bucketsort. Now we pass through this sorted intermediate table, where for each set of tuples that are duplicates w.r.t. their relation parts (they will all be in one contiguous block now that the structure has been sorted), we replace it with one tuple that has this relation part, and a formula part that is the conjunction of the set of formulas corresponding to this set of duplicate tuples. This can also be done in time $O(\|\mathcal{R}\|)$, and thus projection is a linear time ($O(\|\mathcal{R}\|)$ time) operation. ■

Finally, we present the operations intersection and union of two extended relations with the same set of variables. Again we only consider this case because, in our algorithm in the next section, this will always be the case for intersection and union. Note that intersection is a particular case of join.

Definition 5.3.7 (intersection, union). *Let $\mathcal{R}$ and $\mathcal{S}$ be extended X-relations. Then*

1. *the* intersection *of $\mathcal{R}$ and $\mathcal{S}$ is the extended X-relation

$$\mathcal{R} \cap \mathcal{S} = \{(\gamma, \psi) \mid \gamma \in \mathcal{R} \text{ and } \gamma \in \mathcal{S}, \text{ and } \psi = \delta_{\mathcal{R}}(\gamma) \wedge \delta_{\mathcal{S}}(\gamma)\}.$$

2. *the* union *of $\mathcal{R}$ and $\mathcal{S}$ is the extended X-relation

$$\mathcal{R} \cup \mathcal{S} = \{(\gamma, \psi) \mid \gamma \in \mathcal{R} \text{ or } \gamma \in \mathcal{S}, \text{ and } \psi = \delta_{\mathcal{R}}(\gamma) \vee \delta_{\mathcal{S}}(\gamma)\}.$$

As an example of the operations intersection and union, let $\mathcal{R}$ be the extended $\{x, y, z\}$-relation

x	y	z	ψ
1	2	3	A
4	5	6	$\neg C \vee B$
5	7	1	$E \wedge \neg D$

and $\mathcal{S}$ be the extended $\{x, y, z\}$-relation

x	y	z	ψ
2	3	5	$A \wedge C$
4	5	6	$F \vee B$
5	7	1	$\neg F$
6	8	8	$D \wedge E$

The intersection of $\mathcal{R}$ and $\mathcal{S}$, $\mathcal{R} \cap \mathcal{S}$, is the extended $\{x, y, z\}$-relation

x	y	z	ψ
4	5	6	$(\neg C \vee B) \wedge (F \vee B)$
5	7	1	$(E \wedge \neg D) \wedge \neg F$

The union of $\mathcal{R}$ and $\mathcal{S}$, $\mathcal{R} \cup \mathcal{S}$, is the extended $\{x, y, z\}$-relation

x	y	z	ψ
1	2	3	A
2	3	5	$A \wedge C$
4	5	6	$(\neg C \vee B) \vee (F \vee B)$
5	7	1	$(E \wedge \neg D) \vee \neg F$
6	8	8	$D \wedge E$

The following proposition states the semantics of these operations.

Proposition 5.3.8. *Let ϕ_1 and ϕ_2 be formulas with the same set of free variables. Suppose that $\mathcal{R}$ is an answer to ϕ_1 and $\mathcal{S}$ is an answer to ϕ_2, both w.r.t. $\mathcal{A}$. Then*

1. *$\mathcal{R} \cap \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ w.r.t. $\mathcal{A}$;*
2. *$\mathcal{R} \cup \mathcal{S}$ is an answer to $\phi_1 \vee \phi_2$ w.r.t. $\mathcal{A}$.*

Proof: (1) Since the intersection operation is a special case of the join operation where $X = Y$, correctness follows directly from the correctness of the join operation. (2) Since $\mathcal{R}$ is an answer to ϕ_1 w.r.t. $\mathcal{A}$, then for any instantiation γ_1 to the variables X, $\delta_{\mathcal{R}}(\gamma_1)$ is a reduced grounding of $\phi_1[\gamma_1]$. Since $\mathcal{S}$ is an answer to ϕ_2 w.r.t. $\mathcal{A}$, then for any instantiation γ_2 of the variables Y, $\delta_{\mathcal{S}}(\gamma_2)$ is a reduced grounding of $\phi_2[\gamma_2]$. Firstly, $\delta_{\mathcal{R}}(\gamma_1) \vee \delta_{\mathcal{S}}(\gamma_2)$ is a ground formula over $\varepsilon \cup \tilde{A}$, since both $\delta_{\mathcal{R}}(\gamma_1)$ and $\delta_{\mathcal{S}}(\gamma_2)$ are ground formulas over $\varepsilon \cup \tilde{A}$. Secondly, since $\delta_{\mathcal{R}}(\gamma_1)$ is such that for every expansion structure $\mathcal{B}_1 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_1})$ over

$vocab(\phi_1)$, $\mathcal{B}_1 \models \phi_1[\gamma_1] \Leftrightarrow (\mathcal{B}_1, \tilde{A}^{\mathcal{B}_1}) \models \delta_{\mathcal{R}}(\gamma_1)$, and $\delta_{\mathcal{S}}(\gamma_2)$ is such that for every expansion structure $\mathcal{B}_2 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_2})$ over $vocab(\phi_2)$, $\mathcal{B}_2 \models \phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_2, \tilde{A}^{\mathcal{B}_2}) \models \delta_{\mathcal{S}}(\gamma_2)$, it follows that for every expansion structure $\mathcal{B}_3 = (\mathcal{A}; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}_3})$ over $vocab(\phi_1) \cup vocab(\phi_2)$, $\mathcal{B}_3 \models \phi_1[\gamma_1]$ or $\mathcal{B} \models \phi_2[\gamma_2]$ iff $(\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{R}}(\gamma_1)$ or $(\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{S}}(\gamma_2)$. By the semantics of "$\vee$", it follows that for every expansion $\mathcal{B}_3$, $\mathcal{B}_3 \models \phi_1[\gamma_1] \vee \phi_2[\gamma_2] \Leftrightarrow (\mathcal{B}_3, \tilde{A}^{\mathcal{B}_3}) \models \delta_{\mathcal{R}}(\gamma_1) \vee \delta_{\mathcal{S}}(\gamma_2)$, and therefore $\delta_{\mathcal{R}}(\gamma_1) \vee \delta_{\mathcal{S}}(\gamma_2)$ is a reduced grounding of $(\phi_1 \vee \phi_2)[\gamma_3]$ where $\gamma_3 = \gamma_1$ or $\gamma_3 = \gamma_2$. Thus $\mathcal{R} \cup \mathcal{S}$ is an answer to $\phi_1 \vee \phi_2$ w.r.t. $\mathcal{A}$. ∎

The following proposition states the time complexity of any reasonable implementation of the operations intersection and union.

Proposition 5.3.9. *Operations intersection and union have time complexity* $O(\|\mathcal{R}\| + \|\mathcal{S}\|)$.

Proof: Since $\mathcal{R}$ and $\mathcal{S}$ are over a fixed finite domain A, they can be sorted according to some total order on A in time $O(\|\mathcal{R}\|)$ and $O(\|\mathcal{S}\|)$ respectively, using bucket sort. Now the intersection (union) can be computed in a fashion similar to merging two sorted lists, as with join and join with complement. In the case of intersection, this procedure generates, in general, an extended X-relation of size $O(|X||R| + \sum_{\gamma \in \mathcal{R}} \|\delta_{\mathcal{R}}(\gamma)\| + \sum_{\gamma \in \mathcal{S}} \|\delta_{\mathcal{S}}(\gamma)\|) \leq O(\|\mathcal{R}\| + \|\mathcal{S}\|)$. In the case of union, this procedure generates an extended X-relation of size $O(|X| \cdot (|R| + |S|) + \sum_{\gamma \in \mathcal{R}} \|\delta_{\mathcal{R}}(\gamma)\| + \sum_{\gamma \in \mathcal{S}} \|\delta_{\mathcal{S}}(\gamma)\|) \leq O(\|\mathcal{R}\| + \|\mathcal{S}\|)$. ∎

Now that we have defined these five operations for extended relations, we now have a grounding mechanism for model expansion problems, much like the mechanism of relations for solving model checking problems. As such, we can use this representation, the extended relation, and its algebra to develop algorithms for grounding model expansion problems. We present such an algorithm in the next section, which is the main result of this thesis.

Chapter 6

An Algorithm for Grounding First Order Formulas

In the previous section, we presented a mechanism (a representation and an algebra) for producing a reduced grounding of a FO formula over a FO structure. As such, we can now develop recursive algorithms for producing a reduced grounding of a formula over a structure. In this section we present such an algorithm; specifically, we present an algorithm for grounding k-guarded sentences under the restriction that all guards are initially specified (by the structure), the main result of this thesis. Because, like [46] we take advantage of the structural properties of k-guarded formulas, this algorithm runs in time $O(\ell^2 n^k)$. In fact, our algorithm is an extension of this model checking algorithm of [46] for grounding. While Liu and Levesque consider grounding for strictly k-guarded formulas (those formulas in SGF_k), due to the fact that their algorithm is recursively defined, our recursive algorithm considers grounding for a restricted form of strictly k-guarded formulas, (those formulas in the fragment RGF_k).

Definition 6.1 ($\mathbf{RGF}_k$). *RGF_k denotes the set of strictly k-guarded formulas such that no expansion predicate appears in any guard.*

Note that the restriction that "no expansion predicate appears in any guard" is necessary for a polynomial time grounding algorithm. Indeed, there is no polynomial time grounding algorithm even for 1-guarded sentences; otherwise, we would have a polynomial time reduction to SAT for MX for 1-guarded sentences, and hence the combined complexity of this problem would be in **NP**. However, this problem is **NEXP**-complete [37]. Considering only strictly k-guarded formulas is not as restrictive, however, because, as with model checking, for most model expansion problems in practice, the formula is a sentence, and all k-guarded sentences are strictly k-guarded formulas.

It is easy to see that any FO^k formula can be rewritten in linear time into an equivalent one in RGF_k, by using atoms of the form $x = x$ as parts of the guards when necessary. For example, the formula $\exists x \exists y E(x, y)$ can be rewritten into $\exists x \exists y[x = x \land y = y \land E(x, y)]$; and the formula $\exists x \exists y[R(x) \land E(x, y)]$ can be rewritten into $\exists x \exists y[R(x) \land y = y \land E(x, y)]$, where R is an instance predicate, and E is an expansion predicate. Since, by the result of [18] that FO^k has the same expressive power as FO formulas of treewidth at most k, a FO formula with treewidth at most k can be put into an equivalent one in RGF_k. In practice, most formulas have a small treewidth, and thus can be put into RGF_k with small k.

We now present the main result of this thesis, the grounding algorithm. This algorithm is the same as that of [46] except that it uses extended relations and operations on them. Note that, as with Eval, there are two Gnd procedures, but one has two parameters and the other has three. If ϕ is an atomic formula $R(\bar{t})$, we use $\phi(\mathcal{A})$ to denote the extended relation $\{(\gamma, \mathit{true}) \mid \bar{t}[\gamma] \in R^{\mathcal{A}}\}$.

Procedure $\mathrm{Gnd}(\mathcal{A}, \phi)$
Input: A structure $\mathcal{A}$ and a formula $\phi \in \mathrm{RGF}_k$
Output: An answer to ϕ w.r.t. $\mathcal{A}$

Suppose $\phi(\bar{x}) = \exists \bar{y}(G_1 \land \cdots \land G_m \land \psi)$. Return $\pi_{\bar{x}}\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \psi')$, where $\mathcal{R}$ is $G_1(\mathcal{A}) \bowtie \cdots \bowtie G_m(\mathcal{A})$, and ψ' is the result of pushing negation symbols in ψ inward so that they are in front of atoms or existentials.

Procedure $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi)$ is defined recursively by:

1. If ϕ is a positive atom from the instance vocabulary,
   then $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie \phi(\mathcal{A})$;

2. If ϕ is $\neg\phi'$, where ϕ' is an atom from the instance vocabulary,
   then $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \mathcal{R} \bowtie^c \phi'(\mathcal{A})$;

3. If ϕ is an atom from the expansion vocabulary,
   then $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi) = \{(\gamma, \phi[\gamma]) \mid \gamma \in \mathcal{R}\}$;

4. $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, (\phi \land \psi)) = \mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi) \cap \mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \psi)$;

5. $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, (\phi \vee \psi)) = \mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \phi) \cup \mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \psi)$;

6. $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \exists \bar{y}\phi) = \mathcal{R} \bowtie \mathrm{Gnd}(\mathcal{A}, \exists \bar{y}\phi)$;

7. $\mathrm{Gnd}(\mathcal{A}, \mathcal{R}, \neg\exists \bar{y}\phi) = \mathcal{R} \bowtie^c \mathrm{Gnd}(\mathcal{A}, \exists \bar{y}\phi)$.

In the algorithm of [46], every operation except for the join $G_1(\mathcal{A}) \bowtie \cdots \bowtie G_m(\mathcal{A})$ of up to k guards ($m \leq k$) can be done in time linear in size of its input. The algorithm we presented differs from the one of [46] in that it does grounding for model expansion, not model checking. As such, formulas are being copied over in every operation here. For this reason, the running time of Gnd will be greater than that of [46], because join can no longer be done in linear time in its input relations (Proposition 5.3.3).

As an example, let ϕ be the 1-guarded formula $\exists x[R(x) \wedge \exists y(S(y) \wedge E(y))]$, where E is an expansion predicate, and let $\mathcal{A}$ be a structure such that $R^{\mathcal{A}} = \{2i+1 \mid 0 \leq i \leq m\}$, and $S^{\mathcal{A}} = \{2i \mid 0 \leq i \leq m\}$. The algorithm Gnd with input ($\mathcal{A}$, ϕ) takes the following steps in the following order

1. Evaluate guard corresponding to $R(x)$ with respect to $\mathcal{A}$ or $R(\mathcal{A})$, which results in

x	ψ
1	*true*
3	*true*
$\vdots$	$\vdots$
$2m+1$	*true*

Now we need to compute the answer to the subformula $\phi' = \exists y(S(y) \wedge E(y))$ with respect to $\mathcal{A}$ before we go any further, so recursively (steps 2 to 4) we

2. evaluate guard correponding to $S(y)$ with respect to $\mathcal{A}$, or $S(\mathcal{A})$, which results in

y	ψ
0	*true*
2	*true*
$\vdots$	$\vdots$
$2m$	*true*

3. Evaluate $S(\mathcal{A}) \bowtie E(\mathcal{A})$. Since E is an expansion predicate, this results simply in

y	ψ
0	$E(0)$
2	$E(2)$
$\vdots$	$\vdots$
$2m$	$E(2m)$

4. Now we evaulate the $\emptyset$-projection of the table obtained in the previous step, or $\pi_\emptyset(S(\mathcal{A}) \bowtie E(\mathcal{A}))$, which results in

ψ
$\bigvee_{i=0}^{m} E(2i)$

This table is the answer to ϕ' with respect to $\mathcal{A}$ that we want for the next step (we refer to this answer as $\phi'(\mathcal{A})$ for convenience). Now we can compute the remainder of ϕ with respect to $\mathcal{A}$. Next we

5. Evaluate the join of $R(\mathcal{A})$ with $\phi'(\mathcal{A})$, or $R(\mathcal{A}) \bowtie \phi'(\mathcal{A})$, which results in

x	ψ
1	$\bigvee_{i=0}^{m} E(2i)$
3	$\bigvee_{i=0}^{m} E(2i)$
$\vdots$	$\vdots$
$2m+1$	$\bigvee_{i=0}^{m} E(2i)$

6. We then evaluate the $\emptyset$-projection of the table obtained in the previous step, or $\pi_\emptyset(R(\mathcal{A}) \bowtie \phi'(\mathcal{A}))$, which results in the following

ψ
$\bigvee_{i=0}^{m} E(2i)$

So here, the formula $\bigvee_{i=0}^{m} E(2i)$ is the answer to ϕ w.r.t. $\mathcal{A}$, *i.e.*, it is a reduced grounding of ϕ over $\mathcal{A}$, and it has size $O(n)$ (n being the size of $\mathcal{A}$). However, in the computation of ϕ w.r.t. $\mathcal{A}$ by Gnd, the intermediate extended relation computed in step 5, corresponding to the subformula $R(x) \wedge \exists y(S(y) \wedge E(y))$ has size $O(n^2)$. To solve this problem, we let IGnd be the algorithm which is the same as Gnd, except for the following: After each projection operation, we replace each formula in the resulting extended relation by a new propositional symbol. We also save the definitions of these new propositional symbols. The output of the algorithm is the final extended relation together with the definitions of all the new propositional symbols.

For instance, in the above example, in step 4 we will introduce a new propositional symbol p and save the definition $p \equiv \bigvee_{i=0}^{m} E(2i)$. Then the intermediate extended relation computed in step 5, corresponding to the subformula $R(x) \wedge \exists x(S(x) \wedge E(x))$ will be as follows, and it has size $O(n)$

x	ψ
1	p
3	p
$\vdots$	$\vdots$
$2m+1$	p

This modification, in this case, ensures that no intermediate extended relation is larger than $O(n)$ in size. In general, for k-guarded formulas, in Gnd (by the structure of formulas in RGF$_k$), after a projection, the resultant table or its complement is always joined with a guard. This modification ensures that, even though join and join with complement are quadratic time operations (Proposition 5.3.3), the table after a projection always has a form such that this join or join with complement remains of size $O(\ell n^k)$, where ℓ is the size of the formula and n is the size of the structure. It turns out that this is enough to allow the algorithm to still be a polynomial time algorithm in ℓ and n. This is explained in detail in the complexity theorem at the end of this chapter.

Note also, that this modification is to the projection operation only. However, this modification allows the projection operation to still have the same (linear time) complexity, as in Propositon 5.3.6, as the replacement of each formula with propositional symbols in an extended relation requires only one pass of the extended relation.

To illustrate the grounding algorithm IGnd with a full example, let ϕ be the formula

$$\begin{aligned}\exists yzuv.&\underline{R(x,y,z) \wedge S(z,u,v)}\wedge\\ &\neg\{T(x,u,v) \wedge \exists fgh.\underline{A(x,y,f) \wedge B(z,g,h)}\wedge\\ &\qquad[\neg C(f,z) \vee D(x,g) \wedge E(y,h)]\},\end{aligned}$$

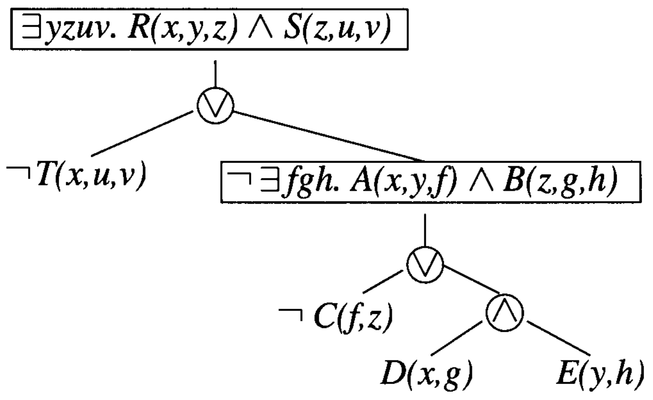

Figure 6.1: RGF_2 formula ϕ' written as a tree

where T and E are expansion predicates. Let $\mathcal{A}$ be the structure with domain $A = \{0, \dots, 9\}$ over vocabulary $\{R, S, A, B, C, D\}$ (note that T and E are not in the vocabulary because they are expansion predicates), where

$R^{\mathcal{A}} = \{(1,2,1), (1,2,2), (7,0,2), (7,0,3)\}$,
$S^{\mathcal{A}} = \{(1,0,2), (2,2,5), (2,3,9), (6,7,8)\}$,
$A^{\mathcal{A}} = \{(1,2,6), (7,0,2), (7,0,5)\}$,
$B^{\mathcal{A}} = \{(1,5,9), (2,3,4), (3,7,5)\}$,
$C^{\mathcal{A}} = \{(1,2), (2,2), (5,2), (6,2), (6,3), (9,6)\}$,
$D^{\mathcal{A}} = \{(1,2), (1,3), (5,7), (7,3), (8,8)\}$.

Then $\phi \in \text{RGF}_2$, where the underlined parts are guards. The grounding algorithm IGnd with input $(\mathcal{A}, \phi)$ takes following steps in the following order. First it pushes negation symbols inward until they are in front of atoms or existentials. Figure 6.1 represents the resulting formula (which we call ϕ') by a tree, where rectangular internal nodes stand for guarded existentials, circular internal nodes for disjunctions or conjunctions, and leaves for literals.

Now the algorithm processes existentials from bottlom to top in the following steps:

1. Evaluate guard corresponding to $R(x,y,z) \wedge S(z,u,v)$ (call it G_1) with respect to $\mathcal{A}$, or $G_1(\mathcal{A}) = R(\mathcal{A}) \bowtie S(\mathcal{A})$. Since $R(\mathcal{A})$ and $S(\mathcal{A})$, the first two tables respectively, are

x	y	z	ψ
1	2	1	*true*
1	2	2	*true*
7	0	2	*true*
7	0	3	*true*

z	u	v	ψ
1	0	2	*true*
2	2	5	*true*
2	3	9	*true*
6	7	8	*true*

x	y	z	u	v	ψ
1	2	1	0	2	*true*
1	2	2	2	5	*true*
1	2	2	3	9	*true*
7	0	2	2	5	*true*
7	0	2	3	9	*true*

then $G_1(\mathcal{A})$ is the above table on the right.

2. Evaluate $G_1(\mathcal{A}) \bowtie^c T(\mathcal{A})$. Since T is an expansion predicate, this results simply in

x	y	z	u	v	ψ
1	2	1	0	2	$\neg T(1,0,2)$
1	2	2	2	5	$\neg T(1,2,5)$
1	2	2	3	9	$\neg T(1,3,9)$
7	0	2	2	5	$\neg T(7,2,5)$
7	0	2	3	9	$\neg T(7,3,9)$

Now we need to compute the answer to the subformula $\phi' = \exists fgh.A(x,y,f) \wedge B(z,g,h) \wedge [\neg C(f,z) \vee D(x,g) \wedge E(y,h)]$ with respect to $\mathcal{A}$ before we go any further, so recursively (steps 3 to 9) we

3. Evaluate guard corresponding to $A(x,y,f) \wedge B(z,g,h)$ (call it G_2) with respect to $\mathcal{A}$, or $G_2(\mathcal{A}) = A(\mathcal{A}) \bowtie B(\mathcal{A})$. Since $A(\mathcal{A})$ and $B(\mathcal{A})$, the first two tables respectively, are

x	y	f	ψ
1	2	6	*true*
7	0	2	*true*
7	0	5	*true*

z	g	h	ψ
1	5	9	*true*
2	3	4	*true*
3	7	5	*true*

x	y	f	z	g	h	ψ
1	2	6	1	5	9	*true*
1	2	6	2	3	4	*true*
1	2	6	3	7	5	*true*
7	0	2	1	5	9	*true*
7	0	2	2	3	4	*true*
7	0	2	3	7	5	*true*
7	0	5	1	5	9	*true*
7	0	5	2	3	4	*true*
7	0	5	3	7	5	*true*

then $G_2(\mathcal{A})$ is the above table on the right.

4. Evaluate $G_2(\mathcal{A}) \bowtie^c C(\mathcal{A})$. Since $C(\mathcal{A})$ is the following table on the left

f	z	ψ
1	2	*true*
2	2	*true*
5	2	*true*
6	2	*true*
6	3	*true*
9	6	*true*

x	y	f	z	g	h	ψ
1	2	6	1	5	9	*true*
7	0	2	1	5	9	*true*
7	0	2	3	7	5	*true*
7	0	5	1	5	9	*true*
7	0	5	3	7	5	*true*

the join of the guard $G_2(\mathcal{A})$ with the complement of $C(\mathcal{A})$ is the above table on the right.

5. Evaluate $G_2(\mathcal{A}) \bowtie D(\mathcal{A})$. Since $D(\mathcal{A})$ is the following table on the left

x	g	ψ
1	2	*true*
1	3	*true*
5	7	*true*
7	3	*true*
8	8	*true*

x	y	f	z	g	h	ψ
1	2	6	2	3	4	*true*
7	0	2	2	3	4	*true*
7	0	5	2	3	4	*true*

the join with the guard $G_2(\mathcal{A})$ is the above table on the right.

6. Evaluate $G_2(\mathcal{A}) \bowtie E(\mathcal{A})$. Since E is an expansion predicate, this results simply in

x	y	f	z	g	h	ψ
1	2	6	1	5	9	$E(2,9)$
1	2	6	2	3	4	$E(2,4)$
1	2	6	3	7	5	$E(2,5)$
7	0	2	1	5	9	$E(0,9)$
7	0	2	2	3	4	$E(0,4)$
7	0	2	3	7	5	$E(0,5)$
7	0	5	1	5	9	$E(0,9)$
7	0	5	2	3	4	$E(0,4)$
7	0	5	3	7	5	$E(0,5)$

7. Now we evaluate the intersection of the tables obtained in steps 5 and 6, or $(G_2(\mathcal{A}) \bowtie D(\mathcal{A})) \cap (G_2(\mathcal{A}) \bowtie E(\mathcal{A}))$, which results in

x	y	f	z	g	h	ψ
1	2	6	2	3	4	$E(2,4)$
7	0	2	2	3	4	$E(0,4)$
7	0	5	2	3	4	$E(0,4)$

8. Now we evaluate the union of the tables obtained in steps 4 and 7, or $(G_2(\mathcal{A}) \bowtie^c C(\mathcal{A})) \cup ((G_2(\mathcal{A}) \bowtie D(\mathcal{A})) \cap (G_2(\mathcal{A}) \bowtie E(\mathcal{A})))$, which results in

x	y	f	z	g	h	ψ
1	2	6	1	5	9	*true*
1	2	6	2	3	4	$E(2,4)$
7	0	2	1	5	9	*true*
7	0	2	2	3	4	$E(0,4)$
7	0	2	3	7	5	*true*
7	0	5	1	5	9	*true*
7	0	5	2	3	4	$E(0,4)$
7	0	5	3	7	5	*true*

9. Now we evaluate the $\{x, y, z\}$-projection of the table obtained in the previous step, or $\pi_{\{x,y,z\}}((G_2(\mathcal{A}) \bowtie^c C(\mathcal{A})) \cup ((G_2(\mathcal{A}) \bowtie D(\mathcal{A})) \cap (G_2(\mathcal{A}) \bowtie E(\mathcal{A}))))$ which results in the following table on the left

x	y	z	ψ
1	2	1	*true*
1	2	2	$E(2,4)$
7	0	1	*true*
7	0	2	$E(0,4)$
7	0	3	*true*

x	y	z	ψ
1	2	1	p_1
1	2	2	p_2
7	0	1	p_3
7	0	2	p_4
7	0	3	p_5

Now, after this projection, in order to keep intermediate tables from becomming larger than $O(\ell n^k)$ in size, we introduce abbreviations for each formula in this table. So here, we replace *true* with p_1, $E(2,4)$ with p_2, *true* with p_3, $E(0,4)$ with p_4, and *true* with p_5. This results in the above table on the right, where we save the definitions p_1 to p_5.

This table is the answer to ϕ' with respect to $\mathcal{A}$ that we want for the next step (we refer to this answer as $\phi'(\mathcal{A})$ for convenience). Now we can compute the remainder of ϕ with respect to $\mathcal{A}$. Next we

10. Evaluate the join of $G_1(\mathcal{A})$ with the complement of $\phi'(\mathcal{A})$, or $G_1(\mathcal{A}) \bowtie^c \phi'(\mathcal{A})$, which results in

x	y	z	u	v	ψ
1	2	1	0	2	$\neg p_1$
1	2	2	2	5	$\neg p_2$
1	2	2	3	9	$\neg p_2$
7	0	2	2	5	$\neg p_4$
7	0	2	3	9	$\neg p_4$

11. Then we evaluate the union of the tables obtained in steps 2 and 10, or $(G_1(\mathcal{A}) \bowtie^c T(\mathcal{A})) \cup (G_1(\mathcal{A}) \bowtie^c \phi'(\mathcal{A}))$, which results in the table

x	y	z	u	v	ψ
1	2	1	0	2	$\neg T(1,0,2) \vee \neg p_1$
1	2	2	2	5	$\neg T(1,2,5) \vee \neg p_2$
1	2	2	3	9	$\neg T(1,3,9) \vee \neg p_2$
7	0	2	2	5	$\neg T(7,2,5) \vee \neg p_4$
7	0	2	3	9	$\neg T(7,3,9) \vee \neg p_4$

12. We then evaluate the $\{x\}$-projection of the table obtained in step 11, or $\pi_{\{x\}}((G_1(\mathcal{A}) \bowtie^c T(\mathcal{A})) \cup (G_1(\mathcal{A}) \bowtie^c \phi'(\mathcal{A})))$, which results in the following table on the left

x	ψ
1	$\neg T(1,0,2) \vee \neg p_1 \vee \neg T(1,2,5) \vee \neg p_2 \vee \neg T(1,3,9)$
7	$\neg T(7,2,5) \vee \neg p_4 \vee \neg T(7,3,9)$

x	ψ
1	q_1
7	q_2

Again, to avoid size blow-up, after this projection, we introduce abbreviations for each formula in the table, that is we replace $\neg T(1,0,2) \vee \neg p_1 \vee \neg T(1,2,5) \vee \neg p_2 \vee \neg T(1,3,9)$ with q_1, and $\neg T(7,2,5) \vee \neg p_4 \vee \neg T(7,3,9)$ with q_2. This results in the above table on the right, where we again save the definitions of q_1 and q_2.

Now that the algorithm is finished, this table resultant of the final step is indeed the answer to the original ϕ with respect to the structure $\mathcal{A}$, as for the two instantiations 1 and 7 of x, the corresponding formulas (namely those represented by q_1 and q_2) are reduced groundings of ϕ with respect to $\mathcal{A}$. That is,

$$\begin{aligned} q_1 &\equiv \neg T(1,0,2) \vee \neg p_1 \vee \neg T(1,2,5) \vee \neg p_2 \vee \neg T(1,3,9) \\ &\equiv \neg T(1,0,2) \vee \mathit{false} \vee \neg T(1,2,5) \vee \neg E(2,4) \vee \neg T(1,3,9) \\ &\equiv \neg T(1,0,2) \vee \neg T(1,2,5) \vee \neg E(2,4) \vee \neg T(1,3,9) \end{aligned}$$

$$\begin{aligned} q_2 &\equiv \neg T(7,2,5) \vee \neg p_4 \vee \neg T(7,3,9) \\ &\equiv \neg T(7,2,5) \vee \neg E(0,4) \vee \neg T(7,3,9) \end{aligned}$$

Both of these are satisfiable, in fact they have many satisfiying assignments as each formula is a disjunction of literals.

Theorem 6.2. *Given a structure $\mathcal{A}$ and a formula $\phi \in RGF_k$, IGnd returns and answer to ϕ w.r.t. $\mathcal{A}$. Hence if ϕ is a sentence, IGnd returns a reduced grounding of ϕ over $\mathcal{A}$.*

Proof: We can rewrite ϕ into an equivalent formula ϕ' as follows: first push negation symbols inward so that they are in front of atoms or existentials (by using DeMorgan's laws and the Double Negation law), then push each guard inward so that it is in front of each of the literals, existentials, or negated existentials that it guards (by using the Distributive law and Idempotence law). Now we can prove by induction that IGnd returns an answer to ϕ'. To prove the base case, consider the conjunction of a guard and a literal. By the definition of IGnd, and the correctness of the join and join with complement operations, it returns an answer to this formula. The induction step follows from the three correctness propositions in the previous chapter. ■

Theorem 6.3. *Given a structure $\mathcal{A}$ and a formula $\phi \in RGF_k$, IGnd runs in time $O(\ell^2 n^k)$, where ℓ is the size of ϕ, and n is size of $\mathcal{A}$.*

Proof: First, we rewrite ϕ into an equivalent RGF_k formula ϕ' by pushing negation symbols inward so that they are in front of atoms or existentials. This takes time $O(\ell)$ as it requires one pass of the formula.

Now we compute the join of all the guards in ϕ'. Since $\phi' \in \mathrm{RGF}_k$, each guard is composed of at most k ordinary relations and thus each join has size $O(n^k)$.

Now, whenever a join $\mathcal{R} \bowtie \mathcal{S}$ of any guard $\mathcal{R}$ with an extended relation $\mathcal{S}$, or the join $\mathcal{R} \bowtie^c \mathcal{S}$ of any guard with the complement of an extended relation $\mathcal{S}$ is performed, it is always the case that the set of variables of $\mathcal{S}$ is a subset of that of $\mathcal{R}$. This follows simply by the fact that ϕ' is k-guarded, that is, $\mathcal{R}$ guards the set of free variables of any subformula (represented by $\mathcal{S}$) below it, and therefore has a set of variables that is a superset of the set of variables in $\mathcal{S}$. Furthermore, it is always the case that every formula in $\mathcal{R}$ is *true* (since $\mathcal{R}$ is a guard composed of ordinary relations), and every formula in $\mathcal{S}$ is of size $O(\ell)$. To see why the latter holds, we consider the three cases. (1) $\mathcal{S}$ represents an atomic subformula. In this case each formula in $\mathcal{S}$ is just *true* and is thus of size $O(\ell)$. (2) $\mathcal{S}$ represents an atomic subformula that is an expansion predicate. In this case, $\mathcal{S}$ is just an extended $\emptyset$-relation composed of this single atomic subformula, which is of size $O(\ell)$. (3) $\mathcal{S}$ represents an existential, (a subformula $\phi'' \in \mathrm{RGF}_k$). Because, with the projection operation, each formula is replaced with a propositional symbol, each formula in $\mathcal{S}$ is just a propositional symbol, and is thus of size $O(\ell)$.

Since each guard is an ordinary relation of size $O(n^k)$, and whenever the join of a guard with an extended relation, or the join of a guard with the complement of an extended relation is performed, the properties described in the above paragraph hold, it follows that any resultant relation will have size $O(\ell n^k)$.

Now that we have exhausted all cases where a join or a join with the complement occurs, the rest of the computation of ϕ over $\mathcal{A}$ involves only the operations projection, union and intersection. These operations are all linear time operations, by the complexity propositions of the previous section. Since there are $O(\ell)$ such operations (the length of the formula is

ℓ), and each intermediate relation has size $O(\ell n^k)$ from the join of each guard with the extended relation representing the subformula below it, it follows that this algorithm runs in time $O(\ell^2 n^k)$ where ℓ is the length of the formula and n is the size of the structure. ■

Chapter 7

Grounding Inductive Definitions

In the previous chapter, we described an algorithm for grounding first order formulas over first order structures that runs in time polynomial in the length of the formula and the size of the structure. This is a good tool for solving model expansion problems, where the formula is a first order formula. MX for FO captures **NP**, *i.e.*, every problem in **NP** can be expressed as an MX problem for some FO formula. However, inductively defined properties like the transitive closure of an edge relation, which are important for modelling applications, are not easy to express in FO logic. For this, we use FO(ID), which is FO augmented with inductive definitions [11, 13], a language which makes the axiomatization of certain problems much more convenient and natural.

The syntax of FO(ID) is that of FO extended with one additional rule stating that an inductive definition (ID) is a formula. That is, FO(ID) is defined by

- If X is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms, then $X(t_1, \ldots, t_n)$ is a formula;
- If Δ is an inductive definition, then Δ is a formula;
- If ϕ and ψ are formulas, then so are $(\neg\phi)$ and $(\phi \wedge \psi)$;
- If ϕ is a formula, then $\exists x\phi$ is a formula, where x is any FO variable.

An inductive definition Δ is a set of *rules* of the form $\forall \bar{x}(X(\bar{t}) \leftarrow \phi)$, where X is a predicate symbol, $\bar{t}$ is a tuple of terms, and ϕ is an arbitrary FO formula. The connective $\leftarrow$ is called the *definitional implication*, and is distinct from material implication, for which we use

$\supset$. A rule $\forall \bar{x}(X(\bar{t}) \leftarrow \phi)$ in a definition does not correspond to the disjunction $\forall \bar{x}(X(\bar{t}) \vee \neg\phi)$ although it implies it. Intuitively, definitional implication should be understood as the "if" found in rules in (informal) inductive definitions, such as "$\neg\phi$ is a formula if ϕ is". In the rule $\forall \bar{x}(X(\bar{t}) \leftarrow \phi)$, $X(\bar{t})$ is called the *head* and ϕ is the *body*. A *defined predicate* of an FO(ID) formula is a predicate symbol that occurs in the head of a rule in an ID; the rest of the predicate symbols are called *open*.

The semantics of FO(ID) is that of FO extended with one additional rule saying that a structure $\mathcal{A}$ satisfies an ID Δ if it is the 2-valued well-founded model of Δ, as defined in the context of logic programming [58]. That is, the semantics of FO(ID) is defined by

- $\mathcal{A} \models X(t_1, \ldots, t_n)$ if $(t_1^{\mathcal{A}}, \ldots, t_n^{\mathcal{A}}) \in X^{\mathcal{A}}$;
- $\mathcal{A} \models \phi \wedge \psi$ if $\mathcal{A} \models \phi$ and $\mathcal{A} \models \psi$;
- $\mathcal{A} \models \neg\phi$ if $\mathcal{A} \not\models \phi$;
- $\mathcal{A} \models \exists x \phi$ if for some value v of x in the domain $dom(\mathcal{A})$ of $\mathcal{A}$, $\mathcal{A}[x : v] \models \phi$;
- $\mathcal{A} \models \Delta$ if $\mathcal{A} = \mathcal{A}^{\Delta\downarrow} = \mathcal{A}^{\Delta\uparrow}$.

More specifically, this extra rule states that a structure $\mathcal{A}$ satisfies an ID Δ if doing induction in two "directions" from two different start points leads to the same (unique) fixpoint. Details of this extra rule can be found in [13].

The model expansion problem for FO(ID) is exactly the same as that for FO except that we consider formulas $\phi \in$ FO(ID), and the semantics of $\models$ is that of FO(ID). As an example, the problem of finding the transitive closure of a graph can be conveniently represented as an MX problem for FO(ID). The formula consists of a defintion with two rules defining the predicate T. The instance vocabulary has a single predicate E, representing the binary edge relation.

$$\left\{ \begin{array}{c} \forall x \forall y [T(x,y) \leftarrow E(x,y)] \\ \forall x \forall y [T(x,y) \leftarrow \exists z (E(x,z) \wedge T(z,y))] \end{array} \right\}$$

The two rules of this ID state that the transitive closure of the set E of edges is the least relation containing all edges and closed under reachability. In this example, the expansion vocabulary includes all the defined predicates. We will assume that this is always the case, in developing methods for solving MX problems for FO(ID).

If we are going to express problems as MX for FO(ID), naturally, we will want to produce groundings of FO(ID) formulas over structures. The notions of *grounding* and *reduced grounding* are analagous to that for FO logic, the only difference is that the semantics of $\models$ is that of FO(ID). The extended relation and its algebra is the mechanism used for grounding a FO formula over a structure. This can be used for grounding a FO(ID) formula over a structure as well, all we need to do is to allow extended relations to contain ground FO(ID) formulas in addition to ground FO formulas. Given this, the notion of an *answer* to a FO(ID) formula with respect to a structure is also analagous to that of FO logic in the same way, and the correctness and complexity of the algebra still applies to this more general form of extended relation. We can thus extend the mechanism we have for grounding a FO formula over a structure to a mechanism for grounding a FO(ID) formula over a structure.

Note, however, that we will not be reducing MX for FO(ID) to SAT, as with FO logic, anymore. Rather, we are reducing MX for FO(ID) to the satisfiability problem of propositional calculus with inductive definitions (PC(ID)). This follows simply from the fact that we allow extended relations to contain ground FO(ID) formulas in addition to FO formulas. As such, the set of reduced groundings represented by any extended relation in this case will be PC(ID) formulas. Currently, two prototypes of such solvers have been developed: one [53] reduces satisfiability of PC(ID) to SAT, and the other [48] is a direct implementation that incorporates SAT techniques. While both solvers deal with a restricted PC(ID) syntax, and are somewhat efficient, a solver for the general syntax (which the output of our grounding algorithm for FO(ID) is part of) is under construction in our lab.

If we want to produce a reduced grounding of an FO(ID) formula over a structure then we always assume that the expansion vocabulary includes all of the defined predicates. Indeed, if defined predicates were in the instance vocabulary, "evaluating them out" of the heads of ground IDs of the formula would produce a PC(ID) formula where $true$ or $false$ occur as heads of rules in its inductive definitions, which is meaningless in PC(ID). Nonetheless, some important problems are represented as MX for FO(ID) where some defined predicates are in the instance vocabulary. The interpretation of these defined predicates is used to apply restrictions on the possible interpretations of the expansion predicates. In such a case, we will first do grounding treating all defined predicates as expansion predicates if they are inside IDs (we still evaluate out any atom of a provided predicate outside of the

IDs as this does not cause a problem). Let ψ be the resulting ground formula. We then add to ψ an extra constraint that encodes the interpretation of the defined predicates. Suppose G is the set of ground atoms of interpreted defined predicates that appear in ψ. Let φ be the conjunction of atoms of G that are true according to the interpretation. Then $\psi \wedge \varphi$ is the final ground formula to be passed to the satisfiability solver. Thus we can restrict our attention to grounding where all defined predicates are expansion predicates.

Since we are able to extend our previous mechanism to one for grounding a FO(ID) formula over a structure, we can now develop recursive algorithms for grounding FO(ID) formulas over a structure, similar to that of Chapter 6. Not surprisingly, we can extend the algorithm of the previous chapter in order to have an efficient algorithm for grounding a FO(ID) formula over a structure. In order to have a polynomial time algorithm for grounding FO(ID) formulas, we consider the fragment IGF$_k$ of FO(ID) formulas

Definition 7.1 (IGF$_k$). *IGF$_k$ is the extension of RGF$_k$ with inductive definitions such that for each rule, the body is in RGF$_k$ and all free variables of the body appear in the head.*

As with RGF$_k$, any FO(ID) formula with at most k distinct variables can be rewritten in linear time into an equivalent one in IGF$_k$. Here we show how to rewrite each rule so that it satisfies the restriction of the above definition. First, for each x that appears in the head but not in the body, add $x = x$ to the body. Now, since the body still uses at most k distinct variables, it can be rewritten into RGF$_k$.

Extending the grounding algorithm of the previous section for grounding IGF$_k$ formulas in polynomial time involves building a procedure to ground inductive definitions in polynomial time, and then simply making this procedure a subroutine of the algorithm of the previous section. Here, we present such a procedure for grounding IDs

Procedure GndID($\mathcal{A}, \Delta$)
Input: A structure $\mathcal{A}$ and an ID $\Delta \in$ IGF$_k$
Output: An answer to Δ w.r.t. $\mathcal{A}$

For each rule r of Δ, suppose r is $\forall \bar{x}(X(\bar{t}) \leftarrow \phi)$, let Δ_r be $\{X(\bar{t})[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \text{IGnd}(\mathcal{A}, \phi)\}$. Return $\bigcup_{r \in \Delta} \Delta_r$.

To illustrate the procedure, let Δ be the ID

$$\left\{ \begin{array}{c} \forall x \forall y [T(x,y) \leftarrow \underline{E(x,y)}] \\ \forall x \forall y [T(x,y) \leftarrow \exists z (\underline{y = y \wedge E(x,z)} \wedge T(z,y))] \end{array} \right\}$$

where T is an expansion predicate. Let $\mathcal{A}$ be over domain $A = \{1,2,3\}$ where $E^{\mathcal{A}} = \{(1,2),(2,3)\}$. Then $\Delta \in \text{IGF}_2$. Note that this ID is the example of the transitive closure of an edge relation given above, only we have added $y = y$ to the body of rule 2 in order for it to be guarded. GndID$(\mathcal{A}, \Delta)$ would proceed as follows

1. Call IGnd$(\mathcal{A}, E(x,y))$ to get extended $\{x,y\}$-relation $E(\mathcal{A})$

x	y	ψ
1	2	*true*
2	3	*true*

   which we will call $\mathcal{R}$

2. Call IGnd$(\mathcal{A}, \exists z(y = y \wedge E(x,z) \wedge T(z,y)))$. This involves first evaluating the guard corresponding to $y = y \wedge E(x,z)$ (call it G_1) with respect to $\mathcal{A}$ or $G_1(\mathcal{A}) = (y = y)(\mathcal{A}) \bowtie E(\mathcal{A})$. Since $(y = y)(\mathcal{A})$ and $E(\mathcal{A})$, the first two tables respectively, are

y	ψ
1	*true*
2	*true*
3	*true*

x	z	ψ
1	2	*true*
2	3	*true*

x	y	z	ψ
1	1	2	*true*
2	1	3	*true*
1	2	2	*true*
2	2	3	*true*
1	3	2	*true*
2	3	3	*true*

   then G_1 is the above table on the right. Then we evaluate $G_1(\mathcal{A}) \bowtie T(\mathcal{A})$. Since T is an expansion predicate, this results in the following table on the left

x	y	z	ψ
1	1	2	$T(2,1)$
2	1	3	$T(3,1)$
1	2	2	$T(2,2)$
2	2	3	$T(3,2)$
1	3	2	$T(2,3)$
2	3	3	$T(3,3)$

x	y	ψ
1	1	$T(2,1)$
2	1	$T(3,1)$
1	2	$T(2,2)$
2	2	$T(3,2)$
1	3	$T(2,3)$
2	3	$T(3,3)$

Then we evaluate the $\{x, y\}$-projection of the previous table, or $\pi_{\{x,y\}} G_1(\mathcal{A}) \bowtie T(\mathcal{A})$, which results in the above table on the right. We will call this $\{x, y\}$-relation $\mathcal{S}$.

3. Replace rule 1 with $\{T(x, y)[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \mathcal{R}\}$, which results in the following set of ground rules

$$T(1,2) \leftarrow true$$
$$T(2,3) \leftarrow true$$

4. Replace rule 2 with $\{T(x, y)[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \mathcal{S}\}$, which results in the following set of ground rules

$$\begin{aligned} T(1,1) &\leftarrow T(2,1) \\ T(2,1) &\leftarrow T(3,1) \\ T(1,2) &\leftarrow T(2,2) \\ T(2,2) &\leftarrow T(3,2) \\ T(1,3) &\leftarrow T(2,3) \\ T(2,3) &\leftarrow T(3,3) \end{aligned}$$

Finally, return the ID that is the union of all the sets of ground rules generated, namely the sets generated in steps 3 and 4. Thus, the resulting ground ID that is output by GndID($\mathcal{A}, \Delta$) is

$$\left\{\begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array}\right\}$$

Here, this ground ID states that the transitive closure T is $\{(1,2),(2,3),(1,3)\}$, which is indeed the least relation containing all edges and closed under reachability with respect to the graph defined by the edge relation $E^{\mathcal{A}} = \{(1,2),(2,3)\}$. Next we formally prove that this algorithm is correct, and give its time complexity.

Lemma 7.2. *Given a structure $\mathcal{A}$ and an inductive definition $\Delta \in IGF_k$, GndID returns an answer to Δ w.r.t. $\mathcal{A}$.*

Proof: Let inductive defintion Δ be such that each rule of Δ is of the form $\forall\bar{x}(X(\bar{t}) \leftarrow \phi)$. Since free variables in the head that are not in body have no effect on the function of a rule of Δ and by the definition of IGF$_k$, w.l.o.g., for each rule of Δ, $\phi \in$ RGF$_k$, where the set of free variables of ϕ is $\bar{x}$. As such, if we call IGnd to get the answer $\mathcal{R}$ to ϕ w.r.t. $\mathcal{A}$, we get a reduced grounding of ϕ for every valuation γ of the tuple $\bar{x}$. That is, by the correctness of IGnd (Theorem 6.0.2) and by the definition of a reduced grounding, for every valuation γ, we have that $\phi[\gamma] \equiv \delta_{\mathcal{R}}(\gamma)$ with respect to the semantics of FO(ID). By the semantics of $\forall$, the rule $\forall\bar{x}(X(\bar{t}) \leftarrow \phi)$ means "for all valuations γ of $\bar{x}$, $X(\bar{t})[\gamma] \leftarrow \phi[\gamma]$". Thus $\forall\bar{x}(X(\bar{t}) \leftarrow \phi)$ is equivalent to $\{X(\bar{t})[\gamma] \leftarrow \phi[\gamma] \mid \gamma \text{ is a valuation of } \bar{x}\}$. This is equivalent to $\{X(\bar{t})[\gamma] \leftarrow \delta_{\mathcal{R}}(\gamma) \mid \gamma \text{ is a valuation of } \bar{x}\}$, which is equivalent to $\{X(\bar{t})[\gamma] \leftarrow \psi \mid (\gamma,\psi) \in \text{IGnd}(\mathcal{A},\phi)\}$ under the semantics of FO(ID). Note that any $\gamma \notin \text{IGnd}(\mathcal{A},\phi)$, its corresponding formula is $false$, so all $X(\bar{t})[\gamma] \leftarrow \psi$ s.t. $\gamma \notin \text{IGnd}(\mathcal{A},\phi)$ is implicitly in the ground ID as well (since it has no affect on the ground ID). Clearly, since Δ is the union of its set of rules, that the union of sets of ground rules equivalent to each rule will also be equivalent to Δ. Thus GndID is correct. ■

Lemma 7.3. *Given a structure $\mathcal{A}$ and an inductive definition $\Delta \in IGF_k$, GndID produces* *a reduced grounding of size $O(\ell n^k)$ and runs in time $O(\ell^2 n^k)$, where ℓ is the size of Δ, and n is the size of $\mathcal{A}$.*

Proof: First run IGnd on the body ϕ of each rule $r \in \Delta$ to produce extended relation $\mathcal{R}$. By the complexity of IGnd (Theorem 6.0.3), this will take time $O(\ell_r^2 n^k)$, where ℓ_r is the length of rule r. Furthermore, since in the run of IGnd, no itermediate extended relation will be larger than $O(\ell_r n^k)$ in size, $\mathcal{R}$ has size $O(\ell_r n^k)$. Now, constructing the set $\{X(\bar{t})[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \text{IGnd}(\mathcal{A}, \phi)\}$ of ground rules for rule r, or $\{X(\bar{t})[\gamma] \leftarrow \delta_{\mathcal{R}}(\gamma) \mid \gamma \in \mathcal{R}\}$ will take time $O(||\mathcal{R}||)$, as it requires one pass of $\mathcal{R}$. This process creates something of size $O(||\mathcal{R}||)$, as the set of ground rules is just the set containing each formula $\delta_{\mathcal{R}}(\gamma)$ of $\mathcal{R}$ with a constant-size header $X(\bar{t})[\gamma] \leftarrow$ appended to the front of it. So, since for each rule r of Δ, it takes time $O(\ell_r^2 n^k)$ to generate its set of ground rules w.r.t. $\mathcal{A}$, and this set of ground rules is of size $O(\ell_r n^k)$, it follows that producing the set of ground rules for all rules of Δ takes time $O(\sum_{r \in \Delta} \ell_r^2 n^k)$, and generates something of size $O(\sum_{r \in \Delta} \ell_r n^k)$. Since $\sum_{r \in \Delta} \ell_r = \ell$, producing the set of ground rules for all rules of Δ takes time $O(\ell^2 n^k)$, and generates something of size $O(\ell n^k)$. Finally, taking the union of all the sets of ground rules for reach rule r of Δ to form the ID that is the output of GndID requires one pass of the sets of ground rules, and thus takes time $O(\ell n^k)$, and generates something of size $O(\ell n^k)$. Thus GndID produces a reduced grounding of size $O(\ell n^k)$ and runs in time $O(\ell^2 n^k)$, where ℓ is the size of Δ, and n is the size of $\mathcal{A}$. ■

Now that we have a subroutine for grounding IDs, we can simply make it a subroutine of the grounding algorithm of the previous section. We do this by adding the following clause to IGnd

0. If ϕ is an ID Δ or its negation $\neg\Delta$, then $\text{IGnd}(\mathcal{A}, \mathcal{R}, \phi) = \{(\gamma, p) \mid \gamma \in \mathcal{R}\}$ or $\{(\gamma, \neg p) \mid \gamma \in \mathcal{R}\}$, where p is a new propositional symbol with the definition $p \equiv \text{GndID}(\mathcal{A}, \Delta)$.

We introduce a new propositional symbol for the result of a call to GndID, for the same reason that we do so for the result of a projection operation, to ensure that each intermediate extended relation is of size $O(\ell n^k)$ in the entire grounding process, where ℓ is the length of the formula, and n is the size of the structure. Adding this rule to IGnd extends this polynomial time grounding algorithm for the fragment RGF_k, to a polynomial time grounding agorithm for the full FO(ID) fragment IGF_k. We call this extended algorithm IGnd^+. To illustrate IGnd^+ with a full example, let ϕ be the FO(ID) formula

$$\exists a\exists c\exists d[\underline{A(a,b)\wedge B(c,d)}\wedge\neg(D(a,c)\vee\left\{\begin{array}{c}\forall x\forall y T(x,y)\leftarrow\underline{E(x,y)}\\ \forall x\forall y T(x,y)\leftarrow\exists z(\underline{y=y\wedge E(x,z)}\wedge T(z,y))\end{array}\right\}\vee$$
$$\neg\exists e\exists f[\underline{F(c,e,f)}\wedge P(c,f)\vee\left\{\begin{array}{c}\forall h N(h)\leftarrow\underline{h=0}\\ \forall h N(h)\leftarrow\exists g(\underline{S(g,h)}\wedge N(g))\end{array}\right\}])]$$

where T, P and N are expansion predicates. let $\mathcal{A}$ be the structure with domain $A=\{0,\ldots,9\}$ over vocabulary $\{A,B,D,E,F,S\}$, where

$A^{\mathcal{A}}=\{(1,2),(3,4),(3,6)\}$
$B^{\mathcal{A}}=\{(5,3),(6,7)\}$
$D^{\mathcal{A}}=\{(1,2),(2,3),(3,5),(5,2),(5,3),(8,4)\}$
$E^{\mathcal{A}}=\{(1,2),(2,3)\}$
$F^{\mathcal{A}}=\{(4,5,7),(5,0,9),(6,3,4),(6,8,7),(9,1,0)\}$
$S^{\mathcal{A}}=\{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9)\}$

Then $\phi\in\mathrm{IGF}_2$, where the underlined parts are guards. First we push negation symbols inward until they are in front of atoms, inductive definitions or existentials. This results in the following equivalent formula

$$\exists a\exists c\exists d[\underline{A(a,b)\wedge B(c,d)}\wedge\neg D(a,c)\wedge\neg\left\{\begin{array}{c}\forall x\forall y T(x,y)\leftarrow\underline{E(x,y)}\\ \forall x\forall y T(x,y)\leftarrow\exists z(\underline{y=y\wedge E(x,z)}\wedge T(z,y))\end{array}\right\}\wedge$$
$$\exists e\exists f[\underline{F(c,e,f)}\wedge P(c,f)\vee\left\{\begin{array}{c}\forall h N(h)\leftarrow\underline{h=0}\\ \forall h N(h)\leftarrow\exists g(\underline{S(g,h)}\wedge N(g))\end{array}\right\}]]$$

Now the algorithm processes the formula from bottom to top. Algorithm IGnd^+ with input $(\mathcal{A},\phi)$ takes the following steps in the following order

1. Evaluate guard corresponding to $A(a,b)\wedge B(c,d)$ (call it G_1) with respect to $\mathcal{A}$, or $G_1(\mathcal{A})=A(\mathcal{A})\bowtie B(\mathcal{A})$. Since $A(\mathcal{A})$ and $B(\mathcal{A})$, the first two tables respectively, are

a	b	ψ
1	2	*true*
3	4	*true*
3	6	*true*

c	d	ψ
5	3	*true*
6	7	*true*

a	b	c	d	ψ
1	2	5	3	*true*
1	2	6	7	*true*
3	4	5	3	*true*
3	4	6	7	*true*
3	6	5	3	*true*
3	6	6	7	*true*

then $G_1(\mathcal{A})$ is the above table on the right.

2. Evaluate $G_1(\mathcal{A}) \bowtie^c D(\mathcal{A})$. Since $D(\mathcal{A})$ is the following table on the left

a	c	ψ
1	2	*true*
2	3	*true*
3	5	*true*
5	2	*true*
5	3	*true*
8	4	*true*

a	b	c	d	ψ
1	2	5	3	*true*
1	2	6	7	*true*
3	4	6	7	*true*
3	6	6	7	*true*

then $G_1(\mathcal{A}) \bowtie^c D(\mathcal{A})$ is the above table on the right. Now we need to compute the join of $G_1(\mathcal{A})$ with the complement of the first ID of the formula ϕ.

3. Call GndID$(\mathcal{A}, \Delta)$, where Δ is

$$\left\{ \begin{array}{c} \forall x \forall y T(x,y) \leftarrow \underline{E(x,y)} \\ \forall x \forall y T(x,y) \leftarrow \exists z(\underline{y = y \wedge E(x,z)} \wedge T(z,y)) \end{array} \right\}$$

Since this ID Δ is the same as in the example given above to illustrate GndID, it produces the following ground ID

$$\left\{ \begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array} \right\}$$

Here we let propositional symbol $p \equiv \text{GndID}(\mathcal{A}, \Delta)$, where we save this definition of p. Since Δ is a negated ID, the join of the result of GndID with the guard is the table

a	b	c	d	ψ
1	2	5	3	$\neg p$
1	2	6	7	$\neg p$
3	4	5	3	$\neg p$
3	4	6	7	$\neg p$
3	6	5	3	$\neg p$
3	6	6	7	$\neg p$

Now we need to compute the answer to the subformula ϕ'

$$\exists e \exists f [\underline{F(c,e,f)} \wedge P(c,f) \vee \left\{ \begin{array}{c} \forall h N(h) \leftarrow \underline{h = 0} \\ \forall h N(h) \leftarrow \exists g (\underline{S(g,h)} \wedge N(g)) \end{array} \right\}]$$

with respect to $\mathcal{A}$, so recursively (steps 4 to 8) we

4. Evaluate the guard corresponding to $F(c,e,f)$ with respect to $\mathcal{A}$, or $F(\mathcal{A})$, which is

c	e	f	ψ
4	5	7	*true*
5	0	9	*true*
6	3	4	*true*
6	7	8	*true*
9	1	0	*true*

5. Evaluate $F(\mathcal{A}) \bowtie P(\mathcal{A})$. Since P is an expansion predicate, this results simply in

c	e	f	ψ
4	5	7	$P(4,7)$
5	0	9	$P(5,9)$
6	3	4	$P(6,4)$
6	7	8	$P(6,8)$
9	1	0	$P(9,0)$

Now we need to compute the join of $F(\mathcal{A})$ with the second ID of ϕ (or the only ID of ϕ').

6. Evaluate $\text{GndID}(\mathcal{A}, \Delta)$ where Δ is

$$\left\{ \begin{array}{c} \forall h N(h) \leftarrow \underline{h = 0} \\ \forall h N(h) \leftarrow \exists g(\underline{S(g,h)} \wedge N(g)) \end{array} \right\}$$

The run of $\text{GndID}(\mathcal{A}, \Delta)$ takes the following steps to give the following solution

(a) Call $\text{IGnd}(\mathcal{A}, h = 0)$ to get $(h = 0)(\mathcal{A})$

h	ψ
0	*true*

which we will call $\mathcal{R}$.

(b) Call $\text{IGnd}(\mathcal{A}, \exists g(S(g,h) \wedge N(g)))$ which involves evaluating guard $S(\mathcal{A})$, which results in the table on the left, then computing $S(\mathcal{A}) \bowtie N(\mathcal{A})$, which results in the table in the middle

g	h	ψ
0	1	*true*
1	2	*true*
2	3	*true*
3	4	*true*
4	5	*true*
5	6	*true*
6	7	*true*
7	8	*true*
8	9	*true*

g	h	ψ
0	1	$N(0)$
1	2	$N(1)$
2	3	$N(2)$
3	4	$N(3)$
4	5	$N(4)$
5	6	$N(5)$
6	7	$N(6)$
7	8	$N(7)$
8	9	$N(8)$

h	ψ
1	$N(0)$
2	$N(1)$
3	$N(2)$
4	$N(3)$
5	$N(4)$
6	$N(5)$
7	$N(6)$
8	$N(7)$
9	$N(8)$

and then computing $\pi_{\{h\}}(S(\mathcal{A}) \bowtie N(\mathcal{A}))$, which results in the above table on the right, which we will call $\mathcal{S}$.

(c) Replace rule 1 with $\{N(h)[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \mathcal{R}\}$, which results in the following set of ground rules

$$N(0) \leftarrow true$$

(d) Replace rule 2 with $\{N(h)[\gamma] \leftarrow \psi \mid (\gamma, \psi) \in \mathcal{S}\}$, which results in the following set of ground rules

$$\begin{gathered} N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{gathered}$$

Thus the output of GndID$(\mathcal{A}, \Delta)$ is

$$\left\{\begin{array}{l} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\}$$

Here we let propositional symbol $q \equiv \text{GndID}(\mathcal{A}, \Delta)$, where we save this definition of q. The join of the result of GndID with the guard is the table

c	e	f	ψ
4	5	7	q
5	0	9	q
6	3	4	q
6	7	8	q
9	1	0	q

7. Evaluate the union of the tables obtained in step 5 and the previous step, or $(F(\mathcal{A}) \bowtie P(\mathcal{A})) \cup \{(\gamma, q) \mid \gamma \in F(\mathcal{A})\}$, which results in

c	e	f	ψ
4	5	7	$P(4,7) \vee q$
5	0	9	$P(5,9) \vee q$
6	3	4	$P(6,4) \vee q$
6	7	8	$P(6,8) \vee q$
9	1	0	$P(9,0) \vee q$

8. Evaluate the $\{c\}$-projection of the table obtained in the previous step, or $\pi_{\{c\}}((F(\mathcal{A}) \bowtie P(\mathcal{A})) \cup \{(\gamma, q) \mid \gamma \in F(\mathcal{A})\})$, which results in the following table on the left

c	ψ
4	$P(4,7) \vee q$
5	$P(5,9) \vee q$
6	$(P(6,4) \vee q) \vee (P(6,8) \vee q)$
9	$P(9,0) \vee q$

c	ψ
4	r_1
5	r_2
6	r_3
9	r_4

After this projection operation, we introduce the abbreviations $r_1 \equiv P(4,7) \vee q$, $r_2 \equiv P(5,9) \vee q$, $r_3 \equiv (P(6,4) \vee q) \vee (P(6,8) \vee q)$ and $r_4 \equiv P(9,0) \vee q$. This results in the above table on the right, where we save the definitions of $r_1, \dots, r_4$. This table is the answer to the subformula ϕ' w.r.t. $\mathcal{A}$, which we call $\phi'(\mathcal{A})$ for convenience.

9. Now we evaluate $G_1(\mathcal{A}) \bowtie \phi'(\mathcal{A})$, which results in

a	b	c	d	ψ
1	2	5	3	r_2
1	2	6	7	r_3
3	4	5	3	r_2
3	4	6	7	r_3
3	6	5	3	r_2
3	6	6	7	r_3

10. Evaluate the intersection of the tables computed in step 3 and the previous step, or $\{(\gamma, \neg p) \mid \gamma \in G_1(\mathcal{A})\} \cap (G_1(\mathcal{A}) \bowtie \phi'(\mathcal{A}))$, which results in

a	b	c	d	ψ
1	2	5	3	$\neg p \wedge r_2$
1	2	6	7	$\neg p \wedge r_3$
3	4	5	3	$\neg p \wedge r_2$
3	4	6	7	$\neg p \wedge r_3$
3	6	5	3	$\neg p \wedge r_2$
3	6	6	7	$\neg p \wedge r_3$

11. Evaluate the intersection of the tables computed in step 2 and the previous step, or $(G_1(\mathcal{A}) \bowtie^c D(\mathcal{A})) \cap (\{(\gamma, \neg p) \mid \gamma \in G_1(\mathcal{A})\} \cap (G_1(\mathcal{A}) \bowtie \phi'(\mathcal{A})))$, which results in

a	b	c	d	ψ
1	2	5	3	$\neg p \wedge r_2$
1	2	6	7	$\neg p \wedge r_3$
3	4	6	7	$\neg p \wedge r_3$
3	6	6	7	$\neg p \wedge r_3$

12. Evaluate the $\{b\}$-projection of the table computed in the previous step, or $\pi_{\{b\}}((G_1(\mathcal{A}) \bowtie^c D(\mathcal{A})) \cap (\{(\gamma, \neg p) \mid \gamma \in G_1(\mathcal{A})\} \cap (G_1(\mathcal{A}) \bowtie \phi'(\mathcal{A}))))$, which results in the following table on the left

b	ψ
2	$(\neg p \wedge r_2) \vee (\neg p \wedge r_3)$
4	$\neg p \wedge r_3$
6	$\neg p \wedge r_3$

b	ψ
2	s_1
4	s_2
6	s_3

After this projection operation, we introduce the abbreviations $s_1 \equiv (\neg p \wedge r_2) \vee (\neg p \wedge r_3)$, $s_2 \equiv \neg p \wedge r_3$ and $s_3 \equiv \neg p \wedge r_3$. This results in the above table on the right, where we save the definitions of s_1, s_2 and s_3.

Now that the algorithm IGnd$^+$ is finished, this table resultant of the final step is indeed the answer to ϕ w.r.t. $\mathcal{A}$, as for the three instantiations 2, 4 and 6 of b, the correponding formulas s_1, s_2 and s_3 are reduced groundings of ϕ w.r.t. $\mathcal{A}$. That is,

$$s_1 \equiv (\neg p \wedge r_2) \wedge (\neg p \wedge r_3) \equiv$$

$$(\neg\left\{\begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array}\right\} \wedge (P(5,9) \vee \left\{\begin{array}{c} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\})) \vee (\neg\left\{\begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array}\right\}$$

$$\wedge((P(6,4) \vee \left\{\begin{array}{c} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\}) \vee (P(6,8) \vee \left\{\begin{array}{c} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\})))$$

$$s_2 \equiv s_3 \equiv \neg p \wedge r_3 \equiv$$

$$\neg\left\{\begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array}\right\} \wedge ((P(6,4) \vee$$

$$\left\{\begin{array}{c} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\}) \vee (P(6,8) \vee \left\{\begin{array}{c} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\}))$$

Both of these are satisfiable because each atom or ID has sets of truth assignments that both satisfy and dissatisfy it, and the rest comes from easy inspection of the above PC(ID) formulas. To see why each ground ID has sets of truth assignments that both satisfy and dissatisfy it, the first ID

$$\left\{\begin{array}{c} T(1,2) \leftarrow true \\ T(2,3) \leftarrow true \\ T(1,1) \leftarrow T(2,1) \\ T(2,1) \leftarrow T(3,1) \\ T(1,2) \leftarrow T(2,2) \\ T(2,2) \leftarrow T(3,2) \\ T(1,3) \leftarrow T(2,3) \\ T(2,3) \leftarrow T(3,3) \end{array}\right\}$$

has the satisfying assignment $\{T(1,2), T(2,3), T(1,3)\}$ (the rest are false), and the dissatisfying assignment $\emptyset$. The second ID

$$\left\{\begin{array}{l} N(0) \leftarrow true \\ N(1) \leftarrow N(0) \\ N(2) \leftarrow N(1) \\ N(3) \leftarrow N(2) \\ N(4) \leftarrow N(3) \\ N(5) \leftarrow N(4) \\ N(6) \leftarrow N(5) \\ N(7) \leftarrow N(6) \\ N(8) \leftarrow N(7) \\ N(9) \leftarrow N(8) \end{array}\right\}$$

has the satisfying assignment $\{N(0), \ldots, N(9)\}$, and the dissatisfying assignment $\{N(0), N(3), N(7)\}$.

Note that if T was provided by the instance structure, for example $T^{\mathcal{A}} = \{(1,2),(2,3),(1,3)\}$ in the above illustration, that we would run the example exactly as we did above, (still treating T as an expansion predicate). But then we would conjoin to each formula s_1, s_2 and s_3, the conjunction $T(1,2) \wedge (2,3) \wedge (1,3)$ (since all of these atoms appear in each of s_1, s_2 and s_3), which encodes the interpretation $T^{\mathcal{A}}$. That is to say, any satisfying assignment to s_1, s_2 or s_3 will have to satisfy $T(1,2) \wedge (2,3) \wedge (1,3)$ and thus takes into account $T^{\mathcal{A}}$. Next, we prove correctness of IGnd$^+$ and give its time complexity.

Theorem 7.4. *Given a structure $\mathcal{A}$ and a FO(ID) formula $\phi \in IGF_k$, IGnd$^+$ returns an answer to ϕ w.r.t. $\mathcal{A}$.*

Proof: We can rewrite ϕ into an equivalent formula ϕ' as follows: first push negation symbols inward so that they are in front of atoms, inductive definitions or existentials (by using DeMorgan's laws and the Double Negation law), then push each guard inward so that it is in front of each of the literals, inductive definitions, negated inductive definitions, existentials, or negated existentials that it guards (by using the Distributive law and Idempotence law). Now we can prove by induction that IGnd returns an answer to ϕ'. We now have two bases cases, the conjunction of a guard with a literal, and the conjunction of a guard with an inductive definition or a negated inductive definition. Correctness of the first base case follows from the defintion of IGnd and the correctness of the join a join with complement operations. Correctness of the second base case follows from the correctness of GndID and the correctness of the join and join with complement operations, since a propositional symbol and a negated propositional symbol are both literals. Given that the base cases are correct, the induction step follows from the three correctness propositions in Chapter 5. ∎

Theorem 7.5. *Given a structure $\mathcal{A}$ and a FO(ID) formula $\phi \in IGF_k$, IGnd$^+$ runs in time $O(\ell^2 n^k)$, where ℓ is the length of the formula, and n is the size of the structure.*

Proof: For each inductive definition $\Delta \in \phi$, run GndID$(\mathcal{A}, \Delta)$ and save this answer as a propositional symbol or a negated propositional symbol, depending on whether Δ is negated or not. By the complexity of GndID, this takes time $O(\ell_\Delta^2 n^k)$ and produces a reduced grounding of size $O(\ell_\Delta n^k)$, where ℓ_Δ is the length of the inductive definition Δ. Since $\sum_{\Delta\in\phi} \ell_\Delta \leq \ell$, this takes time $O(\ell^2 n^k)$, and the set of data that the new propositional symbols represent is of size $O(\ell n^k)$. Since propositional symbols are atomic FO formulas, this procedure has reduced ϕ to a new formula $\phi' \in \mathrm{RGF}_k$. By the complexity of IGnd, we can find the answer to ϕ' w.r.t. $\mathcal{A}$ in time $O(\ell^2 n^k)$, where this answer is of size $O(\ell n^k)$. We now add to this answer the size $O(\ell n^k)$ data set corresponding to the propositional symbols for each ID $\Delta \in \phi$, for an answer to ϕ w.r.t. $\mathcal{A}$ of size $O(\ell n^k)$, which requires one pass of the answer to ϕ w.r.t. $\mathcal{A}$, which takes time $O(\ell n^k)$. Since we do a constant number of time $O(\ell^2 n^k)$ tasks, IGnd$^+$ runs in time $O(\ell^2 n^k)$, where ℓ is the length of the formula, and n is the size of the structure. ∎

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Classical logic is perhaps the most natural language for axiomatizing and solving computational problems. It has a long history, and sophisticated techniques have been developed, which lend themselves to proving correctness of axiomatizations and finding efficient fragments. Classical logic has intuitive and well-understood semantics, convenient syntax, and it is widely used by both theoreticians and practitioners. We therefore believe that tools based on classical logic to the highest degree possible will have a strong appeal.

The model expansion framework of Mitchell and Ternovska [49, 50] is a constraint programming framework based on classical logic extended with inductive definitions. Conceptually, this framework is very close to ASP in that problem instance is given as input (problem description is fixed), and the intention is to find interpretations of unspecified predicates through models of the combined program. However, the approach of the MX framework is to directly work on model expansion without the overhead of logic programming, but rather to rely on classical logic augmented with inductive definitions.

Since ASP systems (and many of the systems presented in Chapter 2) compute a grounding of the problem description over the instance, efficiency of these systems relies on an efficient grounding procedure, and an efficient solver for these ground programs. In our case, since there exists efficient solvers for SAT and PC(ID), an efficient algorithm for grounding an FO or FO(ID) formula over a structure to produce a PC or PC(ID) formula is an asset to solving MX problems efficiently.

This paper represents an important progress in making the MX framework practical by developing an efficient grounding algorithm. In this paper, we have proposed an algorithm for grounding k-guarded FO sentences with inductive definitions under the restriction that all predicates in the guards are initially specified. Our algorithm runs in time $O(\ell^2 n^k)$ where ℓ is the length of the formula, and n is the size of the structure. To this end, we have extended the concept of relation in database theory to the concept of extended relation and defined an algebra on extended relations. As such, we extend the algorithm of Liu and Levesque [46], which uses relational algebra to do model checking, to an algorithm that uses extended relational algebra in order to do grounding. As with [46], the essence of our work is to do efficient grounding by exploiting the structure of FO formulas: Indeed k guarded formulas have the same expressive power as formulas with hypertreewidth at most k.

8.2 Future Work

Current work includes an implementation of this grounding algorithm in the `C` programming language, as part of the MX system. There are several future directions that we can take in regards to grounding. We have shown that taking advantage of the structural properties of the formula is advantageous, however, taking advantage of the structural properties of the instance structure might also allow us to ground efficiently.

While currently this algorithm treats all defined predicates as expansion predicates, many problems in practice include interpretations of the defined predicates in the instance structure. In these cases, symbols of the provided defined symbols will appear in the ground formula. It would preferable if the grounding algorithm could evaluate out these symbols as well. In the case of FO formulas, the symbols of the instance vocabulary are evaluated out during grounding whenever it is determined that a particular ground atom of that vocabulary is $true$ or $false$. This is how it would be approached in evaluating out defined symbols as well. Thus the grounding algorithm would have to understand the semantics of FO(ID) logic, and therefore perform some form of induction during grounding.

Many interesting problems are much more naturally expressed in extensions of FO, such as FO extended with cardinality or weight constraints, as well as FO extended with arithmetic. Since we were able to extend our grounding algorithm for a FO formula, to one for grounding a FO(ID) formula, perhaps we could also devise algorithms for these extensions of FO logic as well.

Also, the techniques used to build an efficient grounding algorithm for FO (with inductive definitions) might also apply to grounding for higher order logics (second order), logics that capture complexity classes larger than **NP**. As the results in other efficient low level solvers appear (*i.e.*, QBF), grounding to SAT or an extension thereof, may not always be necessary. In this case, it would also be interesting to see if similar techniques could be used for grounding second order logic to QBF, for example. We believe that if we take advantage of the structure of the formula in general, that the results will be positive.

Finally, while this grounding algorithm is efficient when the formula is k-guarded, the question arises: if the formula is not given in the k-guarded form, can it be put in this form for k less than the number of variables in the formula. This amounts to computing a hypertree decomposition of the graph associated with the formula of width as small as possible. While it is known that for graphs of bounded treewidth and hypertreewidth, this can be determined in polynomial time, and that a minimal width tree decomposition (hypertree decomposition) can be constructed in polynomial time (in the number of vertices in the graph), there exists a linear time algorithm for determining treewidth, and computing a minimal width tree decomposition. It would be interesting to see if the techniques used here could be applied to hypergraphs, to come up with a recognition (and construction) algorithm that is as efficient as possible. If so, this could be useful to the overall goal of efficiently grounding FO formulas.

Bibliography

[1] Eyal Amir and Sheila McIlraith. Partition-based logical reasoning for first-order and propositional theories. *J. Artif. Intell.*, 162(1-2):49–88, 2005.

[2] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *J. Phil. Logic*, 49(3):217–274, 1998.

[3] Yuliya Babovich and Vladimir Lifschitz. Computing answer sets using program completion. *unpublished draft*, 2003.

[4] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge Univ. Press, 2003.

[5] Kenneth A. Berman, John S. Schlipf, and John V. Franco. Computing well-founded semantics faster. In *Logic Programming and Non-monotonic Reasoning*, pages 113–126, 1995.

[6] D. Berwanger and E. Grädel. Games and model checking for guarded logics. In *Proc. of the 8th Int. Conf. on Logic for Programming and Automated Reasoning, LPAR*, 2001.

[7] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[8] M. Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. In *Proc. of the European Symp. On Programming (ESOP)*, pages 387–401, 2001.

[9] Marco Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. *J. Artif. Intell.*, 162:89–120, 2005.

[10] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proceedings of the 6th International Conference on Database Theory*, volume 1997 of *Lecture Notes in Computer Science*, pages 56–70, 1997.

[11] Marc Denecker. Extending classical logic with inductive definitions. In *Computational Logic - CL 2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 703–717. Springer, 2000.

[12] Marc Denecker. Extending Classical Logic with Inductive Definitions. In *Proc. of the 8th Int. Workshop on Nonmonotonic Reasoning NMR*, pages 1–10, 2000.

[13] Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In *In Proc. of Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 47–60, 2004.

[14] D. East and M. Truszczynski. Predicate-calculus based logics for modeling and solving search problems. *ACM Trans. Comput. Logic*, 2004.

[15] Heinz-Dieter Ebbinghaus and Jorg Flum. *Finite Model Theory.* Springer, 1999.

[16] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proc. of the 7th Int. Workshop on Deductive Databases and Logic Programming (DDLP)*, pages 135–139, 1999.

[17] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Comput.*, pages 43–73, 1974.

[18] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[19] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[20] E. Gonçalvès and E.Grädel. Decidability issues for action guarded logics. In *Proceedings of 2000 International Workshop on Description Logics - DL2000*, pages 123–132, 2000.

[21] Georg Gottlob, Erich Grädel, and Helmut Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Logic*, 3(1):42–79, 2002.

[22] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[23] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: A survey. In *MFCS '01: Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*, pages 37–57, London, UK, 2001. Springer-Verlag.

[24] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. In *Proc. of the 20th ACM-SSS Symp. on Principles of Database Systems (PODS)*, pages 195–206, 2001.

[25] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[26] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.*, 66(4):775–808, 2003.

[27] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[28] E. Grädel. Guarded fragments of first-order logic: a perspective for new description logics? In *Proceedings of 1998 International Workshop on Description Logics DL '98*, Trento, 1998. Extended abstract.

[29] E. Grädel. The decidability of guarded fixed point logic. In J. Gerbrandy, M. Marx, M. de Rijke, and Y. Venema, editors, *JFAK. Essays Dedicated to Johan van Benthem on the Occasion of his 50th Birthday.* Amsterdam University Press, 1999.

[30] E. Grädel. On the restraining power of guards. *J. Symbolic Logic*, 64:1719–1742, 1999.

[31] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002.

[32] E. Grädel, C. Hirsch, and M. Otto. Back and forth between guarded and modal logics. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science LICS 2000*, pages 217–228, Santa Barbara, 2000.

[33] E. Grädel, C. Hirsch, and M. Otto. Back and Forth Between Guarded and Modal Logics. *ACM Transactions on Computational Logics*, 3(3):418 – 463, 2002.

[34] E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science LICS '99, Trento*, pages 45–54, 1999.

[35] N. D. Jones and A. Selman. Turing machines and the spectra of first-order formulae with equality. volume 39, pages 139–150, 1974.

[36] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. In *Proc. of the 17th ACM-SSS Symp. on Principles of Database Systems (PODS)*, pages 205–213, 1998.

[37] A. Kolokolova, Y. Liu, D. Mitchell, and E. Ternovska. Complexity of expanding a finite structure and related tasks, 2006. Presented at the 8th Int. Workshop on Logic and Comput. Complexity (LCC).

[38] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. of the 19th Int. Conf. on Logic Programming (ICLP)*, 2003.

[39] D. Leivant. Descriptive characterizations of computational complexity. *J. Comput. Syst. Sci.*, 39(1):51–83, 1989.

[40] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. Technical Report 1843-02-14, Institut für informationssysteme, TU Wien, 2002.

[41] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving ASP instantiators by join-ordering methods. In *Proc. of the 6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 280–294, 2001.

[42] L. Libkin. *Elements of Finite Model Theory.* Springer, 2004.

[43] Yuliya Lierler and Marco Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Proc. of the 9th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 346–350, 2004.

[44] Vladimir Lifschitz. Computing circumscription. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 121–127. Morgan Kaufmann, 1985.

[45] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *J. Artif. Intell.*, 157:115–137, 2004.

[46] Yongmei Liu and Hector J. Levesque. A tractability result for reasoning with incomplete first-order knowledge bases. In *Proc. of the 18th Int. Joint Conf. on Artif. Intell. (IJCAI)*, pages 83–88, 2003.

[47] V. W. Marek and M. Truszczynski. *Stable logic programming - an alternative logic programming paradigm.* Springer-Verlag, 1999.

[48] Maarten Mariën, Rudradeb Mitra, Marc Denecker, and Maurice Bruynooghe. Satisfiability checking for PC(ID). In *Proc. of Logic for Programming, Artif. Intell., and Reasoning, (LPAR)*, pages 565–579, 2005.

[49] D. G. Mitchell and E. Ternovska. Model extension as a framework for solving NP-Hard search problems, 2005. Presented at the 7th Int. Workshop on Logic and Comput. Complexity (LCC).

[50] David Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc. of the 20th National Conf. on Artif. Intell. (AAAI)*, pages 430–435, 2005.

[51] David G. Mitchell. A SAT solver primer. *Bulletin of the European Assoc.for Theoretical Comput. Sci. (EATCS)*, 85:112–132, 2005.

[52] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artif. Intell.*, 25(3,4):241–273, 1999.

[53] Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In *Proc. of the 21st Int. Conf. on Logic Programming (ICLP)*, pages 221–234, 2005.

[54] Deepak Ramachandran and Eyal Amir. Compact propositional encodings of first-order theories. In *Proc. of the 20th National Conf. on Artif. Intell. (AAAI)*, pages 340–345, 2005.

[55] T. Syrjänen and I. Niemelä. The smodels system. In *Proc. of the 6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 434–438, 2001.

[56] Tommi Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, October 1998.

[57] Tommi Syrjänen. Omega-restricted logic programs. In *Proc. of the 6th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 267–279, 2001.

[58] A. van Gelder, K. A. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1993.

[59] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 82–94, 1981.