Enfragmo: A System for Grounding Extended First-Order Logic to SAT

by

Amir Aavani

M.Sc., Sharif University of Technology, 2006

B.Sc, Iran University of Science and Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Spring, 2014

APPROVAL

Name: Amir Aavani

Degree: Doctor of Philosophy

Title of Thesis: Enfragmo: A System for Grounding Extended First-Order Logic to SAT

Examining Committee:

Dr. Binay Bhattacharya
Chair

Dr. Eugenia Ternovska, Computing Science
Simon Fraser University
Senior Supervisor

Dr. David Mitchell, Computing Science
Simon Fraser University
Supervisor

Dr. James Delgrande, Computing Science
Simon Fraser University
Supervisor

Dr. Ted Kirkpatrick, Computing Science
Simon Fraser University
SFU Examiner

Dr. Marc Denecker, Department of Computing
University of Leuven,
External Examiner

Date Approved: Feb, 18, 2014

Partial Copyright Licence

Abstract

Computationally hard search and optimization problems occur widely in engineering, business, science and logistics, in domains ranging from hardware and software design and verification, to drug design, planning and scheduling. Most of these problems are NP-complete, so no known polynomial-time algorithms exist. Usually, the available solution for a user facing such problems involves mathematical programming for example, integer-linear programming tools, constraint logic programming tools and development of custom-designed implementations of algorithms for solving NP-hard problems. Successful use of these approaches normally requires a deep knowledge of programming, and is often time consuming.

Another approach to attack NP search problems is to utilize the knowledge of users to produce precise descriptions of the (search) problem in a declarative specification or modelling language. A solver then takes a specification, together with an instance of the problem, and produces a solution to the problem, if there is any. Model expansion (MX), the logical task of expanding a given (mathematical) structure by new relations, is one of the well-studied directions of this approach. Formally, in MX, the user axiomatizes their problem in a language. This axiomatization describes the relationship between an instance of the problem (a given finite structure, i.e., a universe together with some relations and functions), and its solutions (certain expansions of that structure).

This thesis presents the Enfragmo system for specifying and solving combinatorial search problems. Enfragmo takes a problem specification, in which the axioms are expressed in an extension of first-order logic, and a problem instance as its input and produces a propositional conjunctive normal form formula that is sent to a propositional satisfiability (SAT) solver. In this thesis, we describe several techniques that we have developed in order to build our well performing solver, Enfragmo.

This thesis is dedicated to
my parents and my wife
for their constant support
and unconditional love.

Drunk, I asked my teacher, "Please, I need to know
What it means to be, or not to be."
He answered me, said, "Go!
Relieve the suffering of the world and you'll be free."
— RUMI'S DIVAN-E SHAMS-E TABRIZI

Acknowledgments

I would like to gratefully thank Dr. Eugenia Ternovska for her guidance, support, and her friendship during my graduate studies at Simon Fraser University. I thank her for having confidence in my abilities to handle such an intricate research topic. Without her critical reviews and intellectual inputs, this thesis would not have been possible in the present form. I would also like to thank my supervisor, Dr. David Mitchell, for his help in developing ideas described in this thesis.

I would like to thank the members of my doctoral committee, Dr. James Delgrane, Dr. Arthur Kikpatrick and Dr. Marc Denecker for their input, valuable discussions and comments.

Finally, I would like to thank my wife Hengameh for her support, encouragement, patience and love. I would like to offer my endless gratitude to my mother and father for their faith in me and teaching me to be as ambitious as I wanted.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many theoretical and real-world problems, we are looking for one or more relations satisfying certain properties. One can name finding a Hamiltonian cycle in a graph, a solution for partially filled Blocked N-Queens problem, or a valid vertex coloring for a graph as theoretical problems having such a nature. Problems, such as how to plan a robot's actions such that it can accomplish a given mission, or how to find the shortest path in a given maze, have more applications in the real-world. These problems, and many other similar problems, are categorized as combinatorial search problems.

One way to solve a search problem is to use the model-based problem solving approach [71]. In this approach, the users needs to describe what the properties of a correct solution are and do not need to worry about how a solution can be found. Users of a model-based problem solver describe their problems and properties of solutions for the solver system in a high-level language, and the system automatically finds solutions for the problems or proves that there is no solution. *Model expansion* based problem solving is a particular kind of model-based problem solving approach. Examples of systems working on this principle are Answer Set Programming (ASP) solvers, such as Clasp [41] and smodels [62], IDP system [72], SPEC2SAT [20] and ESSENSE [39].

One well-studied approach to solving the model expansion problem is to use grounding: Given a problem description in a high-level language, and an instance as the input, *grounding* is the task of translating the input to a low-level language which is called the ground instance. In the next step, the ground instance is fed into an appropriate low-level solver. The solution from the low-level solver will be translated back into a solution to the main problem. Figure 1.1 demonstrates the components in a grounding-based solver.

Grounding-based solvers enable users who are not familiar with programming to solve search problems. These users usually know their problems and the properties of solutions very well. In order to use a specific grounding-based solver, users must simply learn the input language of the solver. The following argument can be used against usefulness of grounding-based solvers for expert users.

Expert users usually have a deep knowledge of the problems they are trying to solve, and they are aware of many heuristics which allow them to find solutions faster. In addition, grounding-based solvers reduce their input to an instance of Satisfiability problem (SAT), for example, and then use a SAT solver to find a solution. These solvers are general purpose, so their grounding algorithms

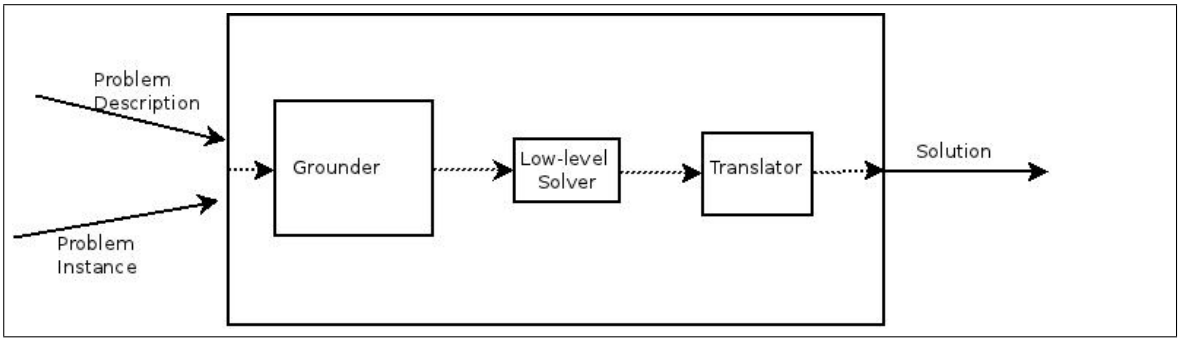

Figure 1.1: Grounding-based Solver: The grounder component receives the problem specification and an instance of that problem, and generates an appropriate reduction for the given problem instance. In the next step, the low-level solver is provided with the result of reduction produced in the previous step. The solution found by the low-level solver is passed to the translator module which extracts the solution.

are generic and are not specialized for any specific problem. Therefore, expert users can develop their own problem-specific programs which generate SAT instances faster than the grounding-based solvers.
This concern can be addressed as follows:

1. Almost all approaches to finding a solution have a corresponding specification in the declarative programming context. Therefore, given an algorithm to solve a problem, an *expert user* can develop a specification which corresponds to that algorithm.
2. Several different reductions can be produced for a given problem, but detecting which one outperforms the others is usually impossible without running some experiments. For almost all interesting problems, comparing the performance of different reductions through the development of specifications and the use of a grounding-based solver is easier than comparing the performance through developing programs.
3. For hard enough problems, e.g., NP-Complete search problems, the solving phase, which is handled by SAT solvers, is the most time consuming element of finding a solution, and the time consumed in order to reduce an instance of the original problem to a SAT instance is usually negligible. Therefore, one cannot expect to receive a huge speed-up by developing a problem-specific grounder.

Since the properties of solutions must be described in the system's input language, the simplicity of the problem description task has a direct effect on how many users will adopt the system. One way to make a solver engine more accessible for both naive and expert users, is to extend the input language of the solver so that it includes commonly used constructs such as functions, aggregates and arithmetical operators.

In this thesis, the Enfragmo system for solving combinatorial search problems is presented. The input language accepted by the Enfragmo system is a rich language based on an extension of first-order logic, augmented with complex terms, such as function and aggregates, and inductive definitions. The low-level solver used by Enfragmo can be any SAT solver. We believe the fact that Enfragmo can be run with any SAT solver is a huge benefit as SAT solvers are improving, every day,

and we can plug-in the most efficient available SAT solver to Enfragmo.

1.1 Enfragmo

Given a specification and a problem instance, Enfragmo solves the problem by reducing its inputs to a SAT instance, and then uses an off-the-shelf SAT solver to find a solution to the generated SAT instance. The solution, if any, will then be translated into a solution to the main problem. Enfragmo generates SAT instances in two phases:

1. **Grounding Phase** Enfragmo generates a variable-free first-order formula which is equivalent to the given problem.
2. **CNF Generation Phase** Enfragmo then converts the variable-free first-order formula to a SAT instance, i.e., a Boolean satisfiability problem instance in the form of Conjunctive Normal Form (CNF).

Enfragmo is equipped with several different approaches for handling of these two phases, the details of which are described in this thesis.

1.2 Summary of contributions

The work described in this thesis is based on joint work with with my supervisors and other coauthors of my papers. The main contributions described in this thesis are:

1. A new problem description language that extends GGF_k logic [65]: Describing the problem in this language is much easier comparing to GGF_k (Chapter 3).
2. Several techniques for writing efficient specifications: These techniques can be used as a guideline to develop specifications for Enfragmo and other model-based solvers (Chapter 3).
3. The relation between GGF_k logic and the input language of Enfragmo: We showed that every problem expressible in GGF_k can also be expressed in the input language of Enfragmo (Chapter 4).
4. Several new algorithms for grounding problems from the high-level language of Enfragmo to a variable-free first-order formula: These algorithms are used by Enfragmo to reduce a problem instance to a variable-free First-Order (FO) formula (Chapter 5).
5. Equipping Enfragmo with approaches for handling specifications with complex terms: We extend the grounding technique introduced in Chapter 5 such that it can handle the rich syntax of Enfragmo, efficiently (Chapter 6).
6. A new approach to generate a CNF from a given variable-free first-order formula: We describe several modifications to the standard Tseitin transformation [67], and also describe a new transformation to generate CNF from a given formula (Chapter 8).

7. Two new propositional encodings for translating cardinality constraints to CNF: We propose two new encodings for translating cardinality constraints to a Satisfiability problem expressed as a Conjunctive Normal Form formula (Chapter 9).

8. A family of approaches for generating a CNF for a given PB-constraint: We propose a family of encodings for translating these constraints to CNF (Chapter 10).

The contributions specific to the author are stated in each chapter. All techniques described in this thesis have been prototyped in Enfragmo.

The structure of this thesis is as follows. In Chapter 2, the technical background and the notations used in the rest of this thesis are set. The language accepted by the solver, Enfragmo, is presented in Chapter 3 and through the use of some examples, we explain how one can express different problems for our solver. Some general techniques proposed by the author of this thesis for developing better (more efficient) problem specifications are also discussed in this chapter. In Chapter 4, the expressive power of the input language of Enfragmo is discussed, and it is demonstrated that every NP search problem that satisfies certain conditions can be expressed in the input language of Enfragmo. In the next two chapters, different methods used by Enfragmo to ground the problems expressed in its input language are described. Several techniques and algorithms for obtaining a variable-free first-order formula from a given specification, in which no aggregates have been used, are described in Chapter 5. In Chapter 6, we explain the algorithms that have been developed to enable Enfragmo to deal with specifications involving complex terms. Chapter 7 describes some of the well-known grounding-based solvers. As we mentioned, Enfragmo passes the intermediate CNF formula to a SAT solver, while all our grounding algorithms generate variable-free first-order formulas for the given problem instance. In Chapter 8, several methods used by Enfragmo to convert a variable-free first-order formula to CNF are reported. Chapters 9 and 10 are dedicated to the description of different approaches to handling Count and Sum aggregates.

Chapter 2

Background

In this chapter, we describe the concept of model expansion for knowledge representation languages without arithmetic operators. We briefly explain the reasons why the same setting cannot be used for the specification languages with built-in arithmetic. In [65], embedded model expansion is introduced for logics with built-in arithmetic. We review embedded model expansion as well.

2.1 Notations

A vocabulary is a set τ of relation and function symbols, each with an associated arity. Constant symbols are function symbols with arity zero. Structure $\mathcal{A}$ for vocabulary τ (or, τ-structure) is a tuple containing a universe, A, and a relation (function) for each relation (function) symbol of τ. For relation symbol R (function symbol f) of the vocabulary, the relation corresponding to R (f) in a τ-structure $\mathcal{A}$ is denoted by $R^{\mathcal{A}}$ ($f^{\mathcal{A}}$, respectively). We write

$$\mathcal{A} = (A; {R_1}^{\mathcal{A}}, \cdots, {R_n}^{\mathcal{A}}, {f_1}^{\mathcal{A}}, \cdots, {f_m}^{\mathcal{A}}),$$

where each R_i is a relation symbol, and each f_i is a function symbol. Throughout this thesis, constants are treated as arity zero functions.

We use $\bar{v} = \langle v_1, \cdots, v_n \rangle$ to represent a tuple of n elements. We say a tuple with n elements has length n, and denote the length of tuple $\bar{v}$ by $|\bar{v}|$. Let $\bar{x} = \langle x_1, \cdots, x_n \rangle$ be a tuple of variables. We use $\phi(\bar{x})$ to denote a formula with free variables $\bar{x}$. To be consistent with the previous work in our group, when we are not referring to formulas, the set of variables occur in $\bar{x}$ is denoted by X, i.e., $X = \{x \mid x \text{ occurs in } \bar{x}\}$.

In this thesis, γ represents an object mapping (assignment). Assignment $\gamma : X \mapsto T$, where X is a set of variables and T is a subset of universe elements, maps each variable in X to an element in T. So, the result of applying assignment γ on tuple of variables $\bar{x}$ is a tuple of elements with length $|\bar{x}|$. Let $\gamma' : X' \mapsto T$ and $\gamma'' : X'' \mapsto T$ be two assignments, and also let X' and X'' be disjoint sets of variables. We use $\gamma' \cup \gamma''$ to denote the assignment whose domain is $X' \cup X''$ and maps $x \in X' \cup X''$ to $t \in T$ iff

- Either $x \in X'$ and $\gamma'(x) = t$,
- Or $x \in X''$ and $\gamma''(x) = t$.

Let Y be a set of variables, T a subset of universe elements, and $\gamma : Y \mapsto T$ an assignment. Also let $\bar{x}$ be a tuple of variables and X be the set of all variables present in $\bar{x}$, such that $Y \supseteq X$. We use $\gamma|_{\bar{x}}$ to denote an assignment from variables in X, to elements in T, which maps $x \in X$ to $\gamma(x)$. In other words, the two assignments, γ and $\gamma|_{\bar{x}}$, agree on the value they assign to all variables in X.

We use $\phi(\gamma|_{\bar{x}})$ to denote formula ϕ applied on the result of instantiation of variables in $\bar{x}$ according to γ. If γ assigns values to all the variables in $\bar{x}$ we may represent $\phi(\gamma|_{\bar{x}})$ by $\phi[\gamma]$. We also use $\phi^{\mathcal{A}}[\gamma]$ to denote the truth value of ϕ in structure $\mathcal{A}$, where structure $\mathcal{A}$ has an interpretation for all relations and functions in $vocab(\phi)$. In this thesis, $vocab(\phi)$ denotes the set of functions and relations present in ϕ.

Similar to formulas, we use $t[\gamma]$ and $f[\gamma]$, where $t(\bar{x})$ is a term, $f(\bar{x})$ is a function, and γ is an assignment to $\bar{x}$, to refer to the result of instantiation of variables in $\bar{x}$ according to γ.

The symbols $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\neg$, $\bot$, $\top$ are used as logical connectives to represent "and", "or", "material implication", "material equivalence", "negation", "*false*", and "*true*", respectively. We use $\Rightarrow$ and $\Leftrightarrow$ as the metalanguage symbols to denote "logical implication" and "logical equivalence", respectively. Also we use $\leftarrow$ as the logical symbol in inductive definitions. In addition, $\Leftrightarrow$ is used as an abbreviation of "if and only if".

2.2 Model Expansion

Model Expansion (MX) formalizes the task behind solving combinatorial search problems. Formally, let σ be a vocabulary, and $\mathcal{A}$ be a structure over σ, $\mathcal{A} = (A; \sigma^{\mathcal{A}})$, where A is the domain (universe) and $\sigma^{\mathcal{A}}$ is an interpretation for all relations in σ. Let $\mathcal{B}$ be a structure whose domain is A with vocabulary $\sigma \cup \epsilon$, i.e., $\mathcal{B} = (A; \sigma^{\mathcal{A}} \cup \epsilon^{\mathcal{B}})$.

Definition 1 (MX) *[54] Let $\mathcal{L}$ be some logic and ϕ be an $\mathcal{L}$-sentence over the union of disjoint vocabularies σ and ε. Given a finite structure $\mathcal{A}$ for vocabulary σ, the model expansion problem $MX(\sigma, \phi)$ asks us to find structure $\mathcal{B}$ which is an expansion of $\mathcal{A}$ to $\sigma \cup \varepsilon$ such that $\mathcal{B} \models \phi$.*[1]

In this thesis, ϕ is called problem specification formula, and is fixed for each search problem, $\mathcal{A}$ always denotes a finite σ-structure, called the instance structure, where σ is the instance vocabulary, and ε the expansion vocabulary.

In the context of model expansion, the problem input is described using the relations and functions in σ, and a solution for the problem is described using the relations and functions in ε. The formula ϕ describes the properties of a correct solution in terms of relations and functions available in the instance vocabulary, and also in terms of relations and functions in the expansion vocabulary.

Consider the following modified version of the graph 3-colouring problem.

Example 1 *Given graph $G = (V, E)$ and three sets of vertices R_p, B_p, G_p, the problem of graph 3-Colouring with Preassigned Colours asks for assigning a colour among Red, Green or Blue to each vertex such that:*

[1] *As we see in the next section, the model expansion was generalized to infinite structures in the embedded setting, in [65].*

1. *If a vertex is in set R_p, it must be coloured Red.*
2. *If a vertex is in set G_p, it must be coloured Blue.*
3. *If a vertex is in set B_p, it must be coloured Green.*
4. *Adjacent vertices must have different colours.*

To express this problem as a model expansion problem, for arbitrary logic $\mathcal{L}$, we can use σ-structure $\mathcal{A} = (A, \sigma^{\mathcal{A}})$ and formula ϕ such that:

- *The domain of the instance structure, A, contains the set of vertices;*
- *σ contains the relations describing the input, i.e., $\{E, R_p, G_p, B_p\}$;*
- *ε contains the relations describing the solution, i.e., $\{R, G, B\}$;*
- *ϕ is any formula in $\mathcal{L}$ describing the above properties. For example, if $\mathcal{L}$ was chosen to be first-order logic, the first property, above, can be expressed using axiom $\forall v : R_p(v) \rightarrow R(v)$.*

Notice that the above "problem specification" is just one of many possible ways to specify graph 3-Colouring with Preassigned Colours.

In this thesis, the focus is on first-order logic model expansion (FO MX). As described in Chapter 4, FO MX can express every problem in NP, but when the problem involves numerical properties, numbers must be encoded by domain elements and arithmetic operations defined using relations over these encodings. There are problems for which expressing numeric properties using encoding for numbers is not straightforward. Having logics with "built-in" arithmetic constructs provides the ability to express the problems more naturally.

Since numbers, in this case integers, are infinite, a structure for a logic in which arithmetic operators have their standard interpretations must have an infinite domain. In model theory, there are two well-studied frameworks for structures with infinite domains:

1. Metafinite Model Theory [44].
2. Embedded Finite Models Theory [50].

In the next section, we describe the embedded model expansion [65], which is an extension of model expansion in the presence of the infinite domain.

2.3 Embedded Model Expansion

Model expansion, explained at the beginning of this chapter, limits the way that arithmetic operations can be used. The reason is that the result of summation of two variables, whose values come from domain A, may not be in A. So the arithmetic operators may not be total. The same issue exists when we consider *aggregate* operators.

Although it is possible to mimic the arithmetic operators by cautious applications of instance functions and predicates, the problem specifications, developed based on this approach, are far from being natural and are usually difficult to understand.

Ternovska and Mitchell proposed the concept of embedded model expansion based on the embedded finite model theory [65]. In the embedded finite model theory, we are dealing with finite relational structures whose elements come from a subset of an infinite domain. In addition, we have access to operations which have a fixed interpretation, e.g., $+$, $\times$.

Definition 2 *([65]) Structure $\mathcal{A}$ is embedded in infinite background (or secondary) structure $\mathcal{M} = (U; \nu)$ if it is a structure $\mathcal{A} = (U; \sigma)$ with finite set σ of finite relations and functions where $\nu \cap \sigma = \emptyset$.*

The set of elements of U that occur in some relation or function of $\mathcal{A}$ is the active domain of $\mathcal{A}$, denoted by $adom_{\mathcal{A}}$ (or $adom$ if $\mathcal{A}$ is clear from the context).

In the context of embedded MX, the following three disjoint vocabularies exist:

1. ν: The vocabulary of background structure $\mathcal{M}$,
2. σ: The vocabulary of instance structure $\mathcal{A}$,
3. ε: The vocabulary of expansion structure $\mathcal{B}$.

Given formula ϕ over vocabulary $\sigma \cup \nu \cup \varepsilon$, and σ-structure $\mathcal{A} = (U, \sigma)$, which is embedded in background structure $\mathcal{M} = (U, \nu)$, embedded MX asks for expanding structure $\mathcal{A}$ to embedded $\sigma \cup \varepsilon$-structure $\mathcal{B}$, such that $\mathcal{B} \models \phi$.

There are four different types of quantifiers, in this setting:

1. Existential quantifier: Formula $\exists x\ \phi(x, \bar{y})$ is equivalent to formula $\bigvee_{a \in U} \phi(a, \bar{y})$.
2. Universal quantifier: Formula $\forall x\ \phi(x, \bar{y})$ is equivalent to formula $\bigwedge_{a \in U} \phi(a, \bar{y})$.
3. Existential quantifier over active domain: Formula $\exists x \in adom\ \phi(x, \bar{y})$ is equivalent to formula $\bigvee_{a \in adom} \phi(a, \bar{y})$.
4. Universal quantifier over active domain: Formula $\forall x \in adom\ \phi(x, \bar{y})$ is equivalent to formula $\bigwedge_{a \in adom} \phi(a, \bar{y})$.

The first two kinds of quantifiers are called generic quantifiers and the last two are called active domain quantifiers. Using the generic quantifiers in logics for MX with infinite background structures can increase the expressive power of logic. In Example 2, the set of even numbers is expressed using an embedded MX specification.

Example 2 *Let $M = (\mathbb{N}, \nu)$, where $\mathbb{N}$ is the set of natural numbers and $+$ operator, with its standard interpretation over natural numbers, is in ν. Also assume $Even$ is a unary expansion predicate. Then the following axiom defines the set of even numbers.*

$$Even(0) \wedge \forall n\ Even(n) \rightarrow (\neg Even(n+1) \wedge Even(n+2))\,.$$

In the above axiom, the $\forall$ quantifier ($\forall n$) is a universal quantifier.

In any structure $\mathcal{B}$ that satisfies the above axiom, $Even^{\mathcal{B}}$ must be equal to the set of even numbers. This example shows that allowing universal quantifiers may result in specifications with no finite ground formula. As long as all quantifiers in a specification are active domain quantifiers, it is guaranteed that there is a finite ground formula for that specification.

In the next section, the concept of expressive power for logics and some other related definitions are presented. In Section 2.5, we describe GGF_k logic, [65], which is a fragment of first-order logic enriched with arithmetic and inductive definitions. The restrictions on GGF_k logic are chosen carefully to limit the expressive power of embedded model expansion on GGF_k to a subset of NP.

2.4 Descriptive Complexity

Descriptive complexity theory is a branch of computational complexity theory which analyzes the expressive power of different logics.

Definition 3 *([55]) Property P of structures is definable in logic $\mathcal{L}$ if there is sentence $\phi \in \mathcal{L}$, such that for every structure $\mathcal{A}$ there is expansion structure $\mathcal{B}$ expanding $\mathcal{A}$ and $\mathcal{B} \models \phi$ iff $\mathcal{A}$ has property P.*

Definition 4 *([55]) Let $\mathcal{D}$ be a class of finite structures, $\mathcal{C}$ be a complexity class, and $\mathcal{L}$ be a logic. Logic $\mathcal{L}$ captures $\mathcal{C}$ on $\mathcal{D}$ if*

1. *For every fixed sentence $\phi \in \mathcal{L}$, the complexity of deciding whether the model expansion problem, defined by formula ϕ and an instance structure from $\mathcal{D}$, is satisfiable, belongs to the complexity class $\mathcal{C}$;*
2. *Every property of structures in $\mathcal{D}$ that can be decided with the complexity $\mathcal{C}$ is definable in logic $\mathcal{L}$.*

The central question of descriptive complexity theory is: "Given a complexity class $\mathcal{C}$, is there a logic that captures $\mathcal{C}$?"

One of the most fundamental results in this field is Fagin's theorem. Theorem 1 states Fagin's theorem in the context of model expansion.

Theorem 1 *Finite model expansion on first-order logic captures NP.*

One of the motivations for introducing model expansion was to study the expressive power of declarative languages which are used in declarative programming solvers. These languages are usually very different from FO logic. The language of Answer Set Programming (ASP), the input language of the IDP system and the input language of Enfragmo are examples of such languages. One of our goals in developing Enfragmo was to capture NP while providing users with a language in which specifying problems is easy.

If arithmetic operators are added to the first-order logic, an infinite set of numbers is required. In Section 2.3, embedded model expansion which provides a framework for supporting arithmetic operators was briefly described.

Unfortunately, in the context of embedded model expansion, having an infinite background structure causes Fagin's theorem to not work. Model expansion on first-order logic, in the presence of infinite background structure and arithmetic operators, can express properties belonging to the complexity classes higher than NP.

Logic GGF_k is designed carefully such that the embedded model expansion on this logic captures a subset of problems in NP. Therefore, having a specification in GGF_k logic and a problem instance, we are certain that we can construct a polynomial time reduction to a SAT problem.

2.5 GGF_k Logic for Embedded Model Expansion

In [65], GGF_k logic that extends the k-guarded fragment of First-Order logic (FO) was introduced. In this logic, instance predicates are used to guard quantifiers and expansion predicates. It has been proven that every NP problem representable by a logical structure that satisfies small cost condition has a corresponding model expansion specification in GGF_k. A structure satisfies small cost condition if the value of the maximum integer in the structure is less than $2^{poly(n)}$, where n is

the number of elements in the domain, and $poly(.)$ is a polynomial. In the rest of this section, the syntax and semantics of GGF_k logic are discussed.

Definition 5 *([65]) The k-guarded fragment, GF_k, of FO is the smallest set of formulas that: 1) contains all atomic formulas; 2) is closed under Boolean operations; 3) contains $\exists \bar{x}\ G_1 \wedge G_2 \cdots \wedge G_m \wedge \phi$, provided the G_i are atomic formulas, $m \leq k$, $\phi \in GF_k$, and each free variable of ϕ appears in some G_i.*

Definition 6 *([65]) The double-guarded fragment $GGF_k(\varepsilon)$ of FO, for a given vocabulary, ε, is the set of formulas of the form $\Phi \wedge \Psi$ with $\varepsilon \subseteq vocab(\Phi \wedge \Psi)$, where Φ is a formula of GF_k and Ψ is a conjunction of guard axioms, one for each symbol of ε, of the form $\forall \bar{x}\ E(\bar{x}) \rightarrow G_1(\bar{x}) \wedge \cdots G_m(\bar{x})$, where $m \leq k$; and the union of free variables in the G_i is precisely $\bar{x}$.*

The guards of $GGF_k(\varepsilon)$ which restrict the range of quantified variables are called lower guards, and the guard axioms of $GGF_k(\varepsilon)$ are called upper guards. $GGF_k(\varepsilon)$ requires all atoms providing upper and lower guards to be from the instance vocabulary, so ranges of variables and expansion predicates are explicitly limited to the active domain of the instance structure.

Definition 7 *([65]) An arithmetical structure is a structure, $\mathcal{N} = (\mathbb{N}; 0, 1, \chi, <, +, ., min, max, \Sigma, \Pi)$ where $\mathbb{N}$ is the set of natural numbers, min, max, Σ and Π are multiset operations, and $\chi[\phi](\bar{x})$ is the characteristic function. Other functions, predicates, and multiset operations may be included, provided every function and relation of $\mathcal{N}$ is poly-time computable.*

The logic for $GGF_k(\varepsilon)$ MX specifications with background structure $\mathcal{N}$ is obtained by extending $GGF_k(\varepsilon)$ with terms constructed from the vocabulary of $\mathcal{N}$.

Definition 8 *([65]) Let τ be the vocabulary $\sigma \cup \nu \cup \varepsilon$ and V a countable set of variables. The set of well-formed terms is the closure of the variables in V and constants in τ under the following operations:*

1. *If f is a function of arity n, other than a multiset operation or the characteristic function, and $\bar{t}$ is a tuple of terms of length n then $f(\bar{t})$ is a term.*
2. *If Γ is a multiset operation[2] of ν, $f(\bar{x}, \bar{y})$ a term, and $\phi(\bar{x}, \bar{y})$ a τ-formula in which $\bar{x}$ is guarded, then $\Gamma_{\bar{x}}((\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}))$, is a term with free variables $\bar{y}$.*
3. *If $\phi(\bar{x})$ is a formula such that $\exists \bar{x} \phi(\bar{x})$ is a k-guarded formula, then $\chi[\phi]$ is a term with free variables $\bar{x}$.*

Definition 9 *([65]) An embedded $GGF_k(\varepsilon)$ MX specification with secondary structure $\mathcal{N}$ is a set of $GGF_k(\varepsilon)$ sentences over $\sigma \cup \varepsilon \cup \nu$, with terms as in Definition 8, and the secondary ν-structure is the arithmetical structure of Definition 7.*

Example 3 *([65]) Here is an embedded MX specification of the KNAPSACK problem (search version). Instance vocabulary $\sigma = (O, w, v, b, k)$, where O is the set of objects, w is the weight*

[2] *Multisets are generalizations of sets that allow multiple occurrence of elements. A multiset operation is a function mapping multisets to values in the range of the operator.*

function, v is the value function, b is the weight bound, k is the value target. Expansion vocabulary $\varepsilon = \{O'\}$, where O' is the set of selected objects. The Background structure is the arithmetical structure $\mathcal{N}$:

- *Axioms:*
  1. $\Sigma_x(w(x) : O(x) \wedge O'(x)) \leq b$,
  2. $k \leq \Sigma_x(v(x) : O(x) \wedge O'(x))$,

  where $\leq$ has its standard interpretation.
- *Upper guard axiom:* $\forall x(O'(x) \rightarrow O(x))$.

The use of the lower guard O, in the axiomatization, corresponds to introducing a type "all objects". In this example, we have used O as the upper guard to enforce $O'(o)$ to be false if we have $\neg O(o)$.

The set of numbers that may occur in a solution for given instance structure $\mathcal{A}$, is restricted to the active domain of $\mathcal{A}$. However, there are problems where this is too restrictive. One way to develop a specification for such a problem in GGF_k logic, is to encode the "new" numbers (those which are not in the active domain) using the numbers in the active domain of $\mathcal{A}$. To relax this limitation, the authors of [65] extended GGF_k logic by "user-defined guard relations". The new logic is called $DGGF_k$ logic.

A specification in $DGGF_k$ has two formulas; D and ϕ. Formula D is over vocabulary $\sigma \cup \delta$, where $\delta \cap \sigma = \emptyset$. Given a σ-structure $\mathcal{A}$, formula D defines an expansion of $\mathcal{A}$ to $\mathcal{A}'$ that includes the interpretation for all the user-defined guard relations, and predicates in δ. The active domain of $\mathcal{A}'$ is the union of $adom_{\mathcal{A}}$ and any elements of the user-defined guard relations. Formula ϕ is over vocabulary $\sigma \cup \delta \cup \nu \cup \varepsilon$, and defines an embedded model expansion task on the $\sigma \cup \delta$-structure $\mathcal{A}'$.

The guard relations in $DGGF_k$ are defined by induction, using the syntax and semantics of FO(ID), the extension of FO with inductive definitions [30].

Chapter 3

Enfragmo: A System for Grounding FO+ to SAT

In this chapter, we present the Enfragmo system and demonstrate the language accepted by Enfragmo, which is an extension of the first-order logic. Enfragmo takes as its input, a problem specification and a problem instance, and it produces a propositional CNF formula that is sent to a SAT solver. The axioms in a problem specification of Enfragmo are formulas in first-order logic enriched with inductive definitions, arithmetical terms, aggregates and functions.

In this chapter, we also demonstrate the role of axiomatization strategies in the system performance, and show experimentally that the performance of Enfragmo is comparable to, if not better than, similar systems.

3.1 Introduction

A major goal of logic programming has been to make programming more declarative. Over the last decade, it has become clear that some computational tasks, such as solving combinatorial search problems, can be effectively handled by purely declarative means. This has been demonstrated, for example, by the increasing use of SAT solvers as core reasoning engines in problem solving and reasoning systems, by the progress in answer set programming, and by related work on constraint modelling languages. In this thesis, we investigate problem solving using the Enfragmo system developed in our group [8]. Our system allows specification of search problems using a purely declarative language. This language is an extension of multi-sorted first-order logic (FO) with arithmetic, aggregate operators, and a limited form of inductive definition.

Given a problem axiomatization and a problem instance, Enfragmo produces a formula of propositional logic in CNF (the grounding phase), and sends it to a SAT solver (the solving phase). If a satisfying assignment is found, a solution for the original problem instance is constructed from it. Formally, the underlying task is model expansion [54], where problem instance is a FO structure, problem specification is a formula, and solutions are expansions of the instance structure that satisfy the specification. In this chapter, the focus is on the specification language of Enfragmo; detail on the techniques and algorithms used in Enfragmo are described in the following chapters.

Many interesting real-world problems cannot be conveniently expressed as a model expansion problem over FO logic without having access to built-in arithmetic operators. Specifically, expressing the problems involving arithmetic or inductive properties as a model expansion in such a logic is not straightforward. Examples of the former include Knapsack and other problems involving weights/costs, while examples of the latter include the Traveling Salesman problem and other problems involving reachability. To help users express their problems, Enfragmo's specification language is enriched with a limited use of inductive definitions, arithmetic and aggregate operators.

Given a rich language like that of Enfragmo, there are many different approaches for representing a problem. It is well-known that different choices of representing a problem can have a large effect on the performance of a SAT solver (or other constraint solving system). In this chapter, we study some techniques which can be used to develop well-performing axiomatization for Enfragmo. These techniques are not exclusive to Enfragmo and can be used/adopted by other logic programming systems.

It is important to mention that these techniques are already known in the problem solving and artificial intelligence areas, specifically in the SAT solving and Integer Programming communities. However, it was not clear that the same kind of performance benefits can be achieved in a high-level language that extends FO logic. Section 3.3 demonstrates that order-of-magnitude performance improvement can be obtained by applying these techniques in developing axiomatizations.

Our experiments confirm that Enfragmo performs well compared to some other logic programming systems. We believe the good performance of our system is due to the following factors:

1. **Separation of grounding and solving:** The grounding algorithms, which are poly-time, are focused on generating good ground representations. Extending the specification language requires only extending the grounding algorithms, not a search algorithm. By grounding to SAT, Enfragmo can always use the latest SAT solvers, or even different solvers for different problems.

2. **Polytime inflationary fix-point induction:** This feature of the specification language allows polytime pre-processing of instances before grounding. The pre-processing is achieved by a purely declarative method, using a set of axioms, and allows for separation of solving of the NP-hard core from polytime computation of useful information about the instance.

3. **Effective axiomatization/modelling strategies:** While it is not possible to predict, in general, which axiomatizations are better, it is important to note that improvements are usually obtained based on axioms that reflect understanding of the problem. That is, developing axiomatizations with good performance is an outcome of understanding the domain, not merely trial-and-error.

The rest of this chapter is organized as follows. In the next section, the syntax in which the problems can be described for Enfragmo are presented. Section 3.3 demonstrates some useful techniques that can be used to specify problems for Enfragmo. These techniques can be applied in other systems which have the appropriate language constructs, although of the exact effect on performance will be system-dependent. Finally, in Section 3.4, we use some experiments to compare the performance of our system and other existing systems. The result confirms our claims about the efficiency of

the techniques described in this chapter and also suggests that, in most cases, our system performs better than the other systems.

3.1.1 My Contributions

Extending the existing syntax with the concepts of Phases and FIXPOINT is contributed by Shahab Tasharrofi and the author. The author proposed the syntax used to express aggregates, functions and arithmetical terms. Also the author is the main developer of the Enfragmo system. The techniques for developing better axiomatizations, presented in Section 3.3, and all the specifications in that section are developed by the author [4].

3.2 Specification Language

In this section, we describe the specification language of the Enfragmo system, which is based on the multi-sorted classical first-order logic extended with inductive definitions, functions, arithmetical and aggregate operators. In Chapter 4, we relate this language to GGF_k and discuss the expressive power of the input language of Enfragmo.

An Enfragmo specification consists of four main sections, delineated by keywords: `GIVEN`, `FIND`, `PHASES` and `PRINT`. The `GIVEN` section defines the sorts (types) and vocabulary used in the specification. The `FIND` section identifies the vocabulary symbols for which the solver must find interpretations, that is, the functions and relations which will constitute a solution (and possibly some "auxiliary relations", which sometimes make axiomatization more convenient). Interpretations of the remaining vocabulary symbols are given by the problem instance. The third section, `PHASES`, consists of one or more `PHASE` sections, and describes the problem axiomatization. Finally, the specifications end with the `PRINT` section which identifies relations that are to be displayed if a solution is found.

Each of the `PHASE` sections contains an optional `FIXPOINT` component, followed by a `SATISFYING` component, which contains a set of sentences of first-order logic. The `FIXPOINT` part, if included, contains an inductive definition, explained in Subsection 3.2.2. The language of the `SATISFYING` part is multi-sorted FO logic with the sorts corresponding to the types defined in the `GIVEN` section, and with arithmetic and aggregates which are described in Subsection 3.2.1. Enfragmo solves phases in the same order as they appeared in the specification and reports unsatisfiability as soon as it cannot find a model for one of the phases. A specification is satisfiable iff all its phases have at least one model. Multiple `PHASE` sections can be used to carry out pre-processing or post-processing that may support more convenient axiomatizations or more efficient solving. Some examples to illustrate how multiple phases can be useful are provided in the next section. We now illustrate the language with examples of specifications.

The input to the system uses an ascii-ized syntax, but we use the more readable "abstract syntax" in our examples here. Full details are included in the "Enfragmo System Manual" [75]. In addition, Appendix E describes the BNF for the input language of Enfragmo.

Example 4 (Graph K-Coloring) *The graph colouring problem is a classic and well-studied NP-hard search problem. The task is to assign a colour from a given set of K colours, to each vertex*

of a graph, so that no two adjacent vertices have the same colour. The following is an Enfragmo specification in which there are two sorts ("types" in the specification language); Vtx and Clr (corresponding to vertices and colours, respectively). We use binary predicate $Edge(u, v)$ to describe the edges of the input graph, and binary predicate $C(v, c)$ to describe the mapping of vertices to colours. In Enfragmo, every sort is ordered, and the $c_2 < c$ in the "at most one" axiom eliminates some symmetries during grounding.

// GC-01: Simple specification for Graph Colouring
GIVEN:
  TYPES: Vtx, Clr*;*
  PREDICATES: $Edge(Vtx, Vtx)$*,* $C(Vtx, Clr)$*;*
FIND: C;
PHASE:
  SATISFYING:
  $\forall v \exists c : C(v, c).$ *//each node has at least one colour*
  $\forall v\ c : C(v, c) \rightarrow \neg\exists c_2 : (c_2 < c) \wedge C(v, c_2).$ *// each node has at most one colour*
  $\forall u\ v\ c : C(u, c) \wedge C(v, c) \rightarrow \neg Edge(u, v).$ *//no two adjacent nodes have same colour*
PRINT: C; *// print the colouring.*

3.2.1 Arithmetic and Aggregates

Arithmetic, including aggregate operators, is formalized as *embedded model expansion over infinite arithmetic structures* with "built-in" predicates, arithmetic and aggregate functions [65], and the input language utilized by Enfragmo uses the same approach. A specification in Enfragmo can have two kinds of types: integer types and enumerated types. Terms of integer types may be used with arithmetic operators, for example, $+$, $-$, $*$, and $ABS(\cdot)$, which have their standard meanings for the integers. Aggregate terms are defined with respect to structure $\mathcal{B}$, in which the formula containing the term is true.

$Max_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M\}$, for any instantiation $\bar{b}$ for $\bar{y}$, denotes the maximum value obtained by $t^{\mathcal{B}}(\bar{a}, \bar{b})$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}(\bar{a}, \bar{b})$ is *true*, or d_M (the default value) if $\phi^{\mathcal{B}}(\bar{a}, \bar{b})$, for all instantiations $\bar{a}$ for $\bar{x}$, is *false*.

$Min_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_m\}$ is defined dually to Max.

$Sum_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y})\}$, for any instantiation $\bar{b}$ for $\bar{y}$, denotes 0 plus the sum of all values $t^{\mathcal{B}}[\bar{a}, \bar{b}]$ for all instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a}, \bar{b}]$ is true.

$Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\}$, for any instantiation $\bar{b}$ for $\bar{y}$, denotes the number of tuples $\bar{a}$ for which $\phi^{\mathcal{B}}[\bar{a}, \bar{b}]$ is true.

As illustrated in Example 5, the syntax of Enfragmo allows nested aggregates.

Example 5 (A Knapsack Problem Variant) *Consider the following variation of the knapsack problem: We are given a set of items (loads) $L = \{l_1, \cdots, l_n\}$, each with an integer weight $W(l_i)$, and m knapsacks $K = \{k_1, \cdots, k_m\}$. The task is to put the n items into the m knapsacks, while satisfying the following constraints. 1) Certain items must go into preassigned knapsacks, as specified by the binary instance predicate, P; 2) H of the m knapsacks have high capacity, and can hold items with total weight Cap_H, while the remainder have capacity Cap_L; 3) No knapsack may contain two items with weights that differ by more than D. Each of Cap_H, Cap_L and D is an instance function*

with arity zero, i.e., a given constant. The following is an Enfragmo specification for this problem where Q is the mapping of items to knapsacks that must be constructed.

GIVEN:
TYPES: $Item, Knaps$*;*
INTTYPES: $Weight, ItemCount$;
PREDICATES: $P(Item, Knapsack), Q(Item, Knapsack)$;
FUNCTIONS:
$W(Item) : Weight,\ Cap_H : Weight,$
$Cap_L : Weight,\ D : Weight, H : ItemCount$;
FIND: Q;
PHASE:
SATISFYING:
// Q is a function mapping items to knapsacks:
$\forall l \exists k : Q(l, k).$
// An item cannot be placed in two different knapsacks:
$\forall l\ k1\ k2 : Q(l, k1) \wedge Q(l, k2) \rightarrow k1 = k2.$
//Q agrees with the pre-assignment P:
$\forall l\ k : P(l, k) \rightarrow Q(l, k).$
// The total weight in each knapsack is at most Cap_H*:*
$\forall k Sum_l\{W(l); Q(l, k)\} \leq Cap_H.$
// At most H knapsacks have total weight greater than Cap_L*:*
$Count_k\{Sum_l\{W(l); Q(l, k)\} > Cap_L\} \leq H.$
// Items in a knapsack differ in weight by at most D:
$\forall k\ l_1\ l_2 : Q(l_1, k) \wedge Q(l_2, k) \rightarrow ABS(W(l_1) - W(l_2)) \leq D.$

3.2.2 Inductive Definitions

Enfragmo supports a limited use of inductive definitions. Definitions are represented in a rule-based syntax where predicates occurring in the heads of the rules are called the defined predicates and all the other predicates are called open. Open predicates in a definition must be instance predicates, that is those interpreted by the structures representing problem instances, or those constructed explicitly in a previous `PHASE` section. Definitions are interpreted under inflationary fixpoint semantics. Well-founded semantics is implemented in Enfragmo as well, but it is not yet efficient. We would like to emphasize that we believe full support of inductive definitions, where definitions are given in terms of "guessed" predicates, is very important. A system supporting such an approach has been implemented by Marc Denecker's group [73]. The IDP system uses MiniSAT(ID) – A SAT solver which is capable of handling inductive definitions natively. But by enabling Enfragmo to use any SAT solver, we followed a different route. However, even the limited form of inductive definitions we implemented, has proven to be very useful in practice. Using them, our system can quickly compute useful information, such as a bound on a predicate or a partial solution, that can be used later to solve the problem more efficiently.

In [59], Pelov and Ternovska proposed a reduction from inductive definitions to the propositional satisfiability problem. By implementing the reduction introduced in [59] we can enable Enfragmo to support well-founded semantics for inductive definitions.

3.3 Axiomatizing for Performance

In this section, we present Enfragmo specifications for a number of NP-hard benchmark problems. To demonstrate strategies for developing axiomatizations which perform well, we present multiple specifications for each problem and compare their performance experimentally. We also demonstrate the value of having a fixpoint operator (polytime inductive definitions) for pre-computing information which an axiomatization can take advantage of to obtain better solving times. Some techniques used, such as reducing symmetries or adding redundant constraints, are common in constraint programming practice, but are not normally used in logic-based knowledge representation. These techniques can be applied in other systems which have the appropriate language constructs, although the effect on performance will be system dependent.

The problems studied in this section are Blocked N-Queens, Graph Colouring, Social Golfers, and Hamiltonian Cycle. These are all widely used NP-hard benchmark problems. For each problem, we describe a specification and a number of variations, each based on an observation about the problem, and we compare the run time of Enfragmo on each of them. The instances are taken from the Asparagus website [1].

In each subsection of this section, we first describe a problem, and then propose several specifications for expressing that problem. Finally, we conclude each subsection with a table comparing the performance of Enfragmo on different specifications. In each table, for each specification, we report the number of instances solved within a 1800 second time-out, the time for producing the ground CNF formula, and the time for the SAT solver to run.

3.3.1 Blocked N-Queen

In this problem, we are given an $N \times N$ board and N queens, and a list of board cells which are "blocked". The task is to place each of the N queens in a non-blocked cell such that no two queens are in the same row, column, or diagonal. The declarations in the specification are as follows:

// Declarations for Blocked N-Queens Specifications
GIVEN:
 TYPES: $Size$;
 PREDICATES: $Blk(Size, Size)$, $Q(Size, Size)$;
FIND: Q;

The remainder of the specification consists of one PHASE with a SATISFYING part only, followed by PRINT Q. Two versions of the SATISFYING part, BQ-01 and BQ-02, are given in Figure 3.1. BQ-01 is, arguably, the most straightforward specification of the problem. BQ-02 uses the Count aggregate to express all "at-most-one" constraints.

Our experience is that using Count aggregate in this way often produces a better performance than the "direct" method used in BQ-01. This is interesting in light of the general intuition that specifications of counting-based properties in CNF are not well handled by SAT solvers. The method of reducing these constraints to CNF is important to the performance, of course. As demonstrated in Chapter 9, Enfragmo implements several methods that are user-selectable. The default, used here, is a CNF formula that models a divide-and-conquer computation of the count. The choice of this default is justified by experience, and experiments reported in [6].

	// BQ-02 SATISFYING part.
	// Using Count aggregate.
	// No queen in a blocked cell
// BQ-01 SATISFYING part.	$\forall r, c : Blk(r, c) \rightarrow \neg Q(r, c).$
// No queen in a blocked cell	// Exactly n queens on the board
$\forall\, r, c : Blk(r, c) \rightarrow \neg Q(r, c).$	$Count_{r,c}\{Q(r, c)\} = N.$
// At least one queen in each row	// At most one queen in each row
$\forall\, r \exists c : Q(r, c).$	$\forall r : Count_c\{Q(r, c)\} \leq 1.$
// At most one queen in each row	// At most one queen in each column
$\forall\, r, c_1, c_2 : (c_1 < c_2 \wedge Q(r, c_1)) \rightarrow \neg Q(r, c_2).$	$\forall c : Count_r\{Q(r, c)\} \leq 1.$
// At most one queen in each column	// At most one queen in each diagonal
$\forall\, c, r_1, r_2 : (r_1 < r_2 \wedge Q(r_1, c)) \rightarrow \neg Q(r_2, c).$	$\forall d : Count_{r,c}\{Q(r, c) \wedge r - c = d - 1\} \leq 1.$
// At most one queen in each diagonal	$\forall d : Count_{r,c}\{Q(r, c) \wedge c - r = d\} \leq 1.$
$\forall\, r, c, d : Q(r, c) \rightarrow \neg Q(r + d, c + d).$	$\forall d : Count_{r,c}\{Q(r, c) \wedge c + r = 2d\} \leq 1.$
$\forall\, r, c, d : Q(r, c) \rightarrow \neg Q(r + d, c - d).$	$\forall d : Count_{r,c}\{Q(r, c) \wedge c + r = 2d - 1\} \leq 1.$

Figure 3.1: Specifications BQ-01 and BQ-02 for Blocked N-Queens.

Another improvement can be obtained by observing that the complement of relation Blk (blocked) is an upper bound for the relation Q (queen): If $Q(r, c)$ then $\neg Blk(r, c)$. We can rewrite the axioms by guarding $Q(r, c)$ with its upper bound, $\neg Blk(r, c)$, and get a logically equivalent specification. For example, the first of the axioms for "at most one queen in each diagonal" of specification BQ-2 becomes

$$\forall\, d : Count_{r,c}\{\neg Blk(r, c) \wedge Q(r, c) \wedge (r - c) = d\} \leq 1.$$

We call the specification obtained by adding upper bounds to the expansion predicates, in all axioms of BQ-01 (BQ-02), specification BQ-03 (BQ-04, respectively). The solving time of BQ-03 (BQ-04) is smaller than that of BQ-01 (BQ-02, respectively), but at the cost of a slightly longer grounding time. We observe a similar trade-off by applying the same technique to many other problems. A general, automated technique for deriving bounds is reported in [74].

Specification	Axioms	Number Solved	Grounding Time	Solving Time
BQ-01	FO	258/400	0.13	323.8
BQ-02	Count	397/400	4.4	38.4
BQ-03	FO+Upper-Guards	264/400	0.09	269.2
BQ-04	Count+Upper-Guards	400/400	4.2	24.4

Table 3.1: Mean running times, in seconds, for different specifications of Blocked N-Queens. The best performance, shown in bold, is obtained by the specification BQ-04.

3.3.2 Graph k-Colouring

This problem asks if there exists a mapping of vertices of a given graph to colours from a given set of K colours, such that no two adjacent vertices have the same colour. To axiomatize this problem, we use two sorts, Vtx and Clr, corresponding to vertices in the graph and available colours, respectively. We assume the graph is described using binary predicate $Edge$, and we are asked to return a binary relation C, satisfying a proper colouring. One simple specification for graph K-colouring was given in Example 4. We call that specification GC-01.

We can also use Count to express the constraints that every vertex must have exactly one colour, and every edge is monochromatic:

// every vertex has exactly one colour
$\forall v : Count_c\{C(v, c)\} = 1.$
// for every colour, there is no edge whose both vertices have the same colour
$\forall c : Count_{u,v}\{C(u, c) \wedge C(v, c) \wedge Edge(u, v)\} = 0.$

Graph colouring can also be specified in a quite different manner, using binary encodings of colours, as described in [70]. Given K colours and graph G with m edges, let $L = \lceil \lg(K) \rceil$. The colours are integers between $0, \cdots, K-1$, written as L-bit binary numbers. We write $Col(v, b)$ if the b-th bit of the colour assigned to vertex v is one. If there is an edge between two vertices, there must be at least one place at which the binary representations of the colours assigned to those two vertices are different:

$$\forall u, v : Edge(u, v) \rightarrow \exists b : (Col(u, b) \wedge \neg Col(v, b)) \vee (\neg Col(u, b) \wedge Col(v, b)).$$

This axiom, alone, describes the graph $2^{\lceil \lg(K) \rceil}$-colouring problem. Let $KRep(b)$ be true iff the b-th bit of $K-1$, in binary representation, is one. We may ensure the colour assigned to vertex v, is certainly less than K with:

$$\forall b_1, v : (\neg KRep(b_1) \wedge (\forall b_2 > b_1 : KRep(b_2) \rightarrow Col(v, b_2))) \rightarrow \neg Col(v, b_1).$$

Figure 3.2 illustrates a specification in this manner. It uses a `FIXPOINT` part, in which we use polynomial time computable inductive definitions, to perform the transformation of binary encodings of colours to numbers. Predicate T is an auxiliary predicate defining the relationship between the $Col(v, .)$, binary representation of the colour assigned to vertex v, and $C(v, .)$, the colour assigned to vertex v.

Often, reducing symmetries in the problems search space improves the performance of solvers [17]. To achieve this aim, we can add some axioms to break the symmetries.

Observation 1 *Let $v_1, \cdots, v_l$ induce a clique of size l in graph G, $c_1, \cdots, c_l$ be l distinct colours, and $l \leq k$, then the following are equivalent:*

1. *G has a proper k-colouring.*
2. *G has a proper k-colouring in which each v_i gets colour c_i, $1 \leq i \leq l$.*

So, if we find a clique in G, we may fix the colours given to its vertices. Figure 3.2 shows the use of a set of inductive definitions as a FIXPOINT section in an Enfragmo specification, to construct

GIVEN:
TYPES: Vtx, Bit, Clr;
PREDICATES:
$Edge(Vtx, Vtx), K(Bit)$,
$T(Vtx, Bit, Clr)$,
$C(Vtx, Clr)$,
$Col(Vtx, Bit)$;
FIND: Col;
PHASE:
SATISFYING:

$$\forall u, v : Edge(u, v) \rightarrow \exists b : Col(u, b) \wedge \neg Col(v, b) \vee (\neg Col(u, b) \wedge Col(v, b)).$$

$$\forall v, b_1 : \neg KRep(b_1) \wedge (\forall b_2 > b_1 : (KRep(b_2) \rightarrow Col(v, b_2))) \rightarrow \neg Col(v, b_1).$$

PHASE:
FIXPOINT: (T, C)

$$T(v, i, c) \leftarrow \neg Col(v, 1) \wedge i = MAX_{Bit} \wedge c = 0.$$

$$T(v, i, c) \leftarrow Col(v, 1) \wedge i = MAX_{Bit} \wedge c = 1.$$

$$T(v, i, c) \leftarrow \neg Col(v, i) \wedge \exists c_2 : c = 2c_2 \wedge T(v, i + 1, c_2).$$

$$T(v, i, c) \leftarrow Col(v, i) \wedge \exists c_2 : c = 2c_2 + 1 \wedge T(v, i + 1, c_2).$$

$$C(v, c) \leftarrow T(v, 0, c).$$

PRINT: C

GIVEN:
$\cdots$
PREDICATES:
$Edge(Vtx, Vtx), Col(Vtx, Clr)$,
$Deg(Vtx, Vtx), Clique(Vtx)$,
$\cdots$
PHASE:
FIXPOINT: (Deg, Clique)

$$Deg(v) = k \leftarrow (k = Count_u\{Edge(u, v)\}).$$

$$InClq(u) \leftarrow \forall v \neq u : (Deg(v) < Deg(u)) \vee (Deg(u) = Deg(v) \wedge u < v).$$

$$InClq(u) \leftarrow (\forall v : InClq(v) \rightarrow Edge(u, v)) \wedge (\forall v \neq u : \neg InClq(v) \rightarrow (Deg(v) < Deg(u)) \vee (Deg(u) = Deg(v) \wedge u < v)$$

$\cdots$
PHASE:
SATISFYING:
$\cdots$

$$\forall v : InClq(v) \rightarrow (Col(v, Count_u\{InClq(u) \wedge u \leq v\})).$$

Figure 3.2: Specification techniques for Colouring: The Binary encoding specification on the left, and Clique computation on the right.

a maximal clique in G, which can be coloured immediately. This set of axioms is correct only for graphs with no isolated vertex, but it can be modified, slightly, to work in general.

Experimental results for Graph Colouring specifications are given in Table 3.2. GC-01 is the basic FO specification; GC-02 is the binary encoding version with clique computation; GC-03 is the basic FO specification with clique computation, and GC-04 combines Count-based specification with clique computation.

An interesting sidenote is that we tried to develop a specification for IDP which asserts the same set of facts as GC-04 does, i.e., finding a maximal clique and forcing a colour to each vertex. While Enfragmo solved all 106 instances of graph-colouring using GC-04, the performance of IDP on a specification equivalent to GC-04 was worse than its performance on a naive specification, like GC-01. The inductive definitions supported by IDP are more expressive than those supported by Enfragmo, so it is possible to write a logically equivalent specification and, in principle, in both

Specification	Axioms	Number Solved	Grounding Time	Solving Time
GC-01	FO	$73/106$	0.14	360.0
GC-02	Bin + Clique	$90/106$	0.34	434.4
GC-03	FO + Clique	$100/106$	0.11	278.7
GC-04	Count + Clique	$106/106$	0.12	211.0

Table 3.2: Comparison of the performance of different specifications of the same problem. The best result, shown in bold, is obtained by GC-04.

cases only a polytime computation is required to compute the defined predicates. However, the semantics of the languages for definitions differ: inflationary semantics for Enfragmo and well-founded semantics for IDP. It seems that, in this case, the definition for Enfragmo leads to efficient computation, while those we have tried for IDP do not. It is reasonable to expect that the reverse pattern will occur in some other situations.

3.3.3 Social Golfers

The Social Golfer problem describes a scenario where $g \times s$ golfers are scheduled into into g groups of s players over w weeks, such that every two golfers play in the same group exactly once. We know that for appropriately selected values of g, s and w, the problem always has a solution. The problem instance specifies the number of golfers, groups, weeks and the size of each group, in unary format. The size of each group is equal to the number of golfers divided by number of groups. We use three sorts, $Golfers$, $Groups$ and $Weeks$. Predicate $M(w, g, p)$ states that golfer p plays in group g at week w. Any interpretation for M that satisfies the following axioms is a solution for our problem and vice versa.

1. Each group must have $GroupSize$ golfers in it: $\forall w, g : Count_p\{M(w, g, p)\} = GroupSize$,
2. No two golfers can play in the same group more than once:
   $\forall p_1, p_2 : (p_1 < p_2) \rightarrow Count_{w,g}\{M(w, g, p_1) \wedge M(w, g, p_2)\} \leq 1$,
3. In each week, all golfers must play in exactly one group: $\forall w, p : Count_g\{M(w, g, p)\} = 1$.

These axioms are the only axioms in the `SATISFYING` part of specification SG-01. The search space of the social golfer problem contains several symmetries. The following "symmetry breaking axioms" can be used to break some of these symmetries:

1. In each week, golfer number one is going to play in group number one: $\forall w : M(w, 1, 1)$,
2. In each week, groups are ordered based on the least index of golfers who are playing in the groups: $\forall w, g_1, g_2 : Succ(g_1, g_2) \rightarrow Min_p\{p; M(w, g_1, p); 0\} < Min_p\{p; M(w, g_2, p); 0\}$,
3. The second smallest index of the golfers in the first group of week w_1 must be less than or equal to the second smallest index of golfers in the first group of week w_2, for all $w_2 > w_1$: $\forall w_1, w_2 : Succ(w_1, w_2) \rightarrow Min_p\{p; M(w_1, 1, p) \wedge p \neq 1; 0) < Min_p\{p; M(w_2, 1, p) \wedge p \neq 1; 0\}$.

In the above axioms, the default values used in Min aggregates are not important and can be any arbitrary number.

Specification	Axioms	Number Solved	Grounding Time	Solving Time
SG-01	Basic	126/175	0.0	53.2
SG-02	Basic + Symmetry Ax. 1,2	149/175	0.02	24.8
SG-03	Basic + Symmetry Ax. 1-3	161/175	0.03	19.8

Table 3.3: Comparison of the performance of different specifications of the same problem. The best result, shown in bold, is obtained by SG-03.

3.3.4 Hamiltonian Path

A Hamiltonian Path in directed graph G is a path visiting all vertices of G exactly once. An instance for this problem is a directed graph, described using sort $Nodes$ and predicate $Edge$, and a vertex which is the starting node of the Hamiltonian path, described by function $Start$.

We define three predicates, $P(u, v)$, $Last(l)$, and $R(v, d)$, where $P(u, v)$ is true iff the edge (u, v) is part of the Hamiltonian path, $Last(l)$ is true iff l is the last vertex in the path we are constructing, and $R(v, d)$ is true iff vertex v is the d-th node in the path. The following axioms represent the Hamiltonian path problem:

1. There is a unique last element: $Count_l\{Last(l)\} = 1$,
2. Every edge in the path is also an edge in G: $\forall u, v : P(u, v) \rightarrow Edge(u, v)$,
3. Vertex $Start$ is the starting node of the path: $\forall v : \neg P(v, Start)$,
4. There is no edge in the path which leaves the last node of the path: $\forall v, l : Last(l) \rightarrow \neg P(l, v)$,
5. Every vertex is visited exactly once: $\forall v : (Count_u\{P(u, v)\} = 1 \vee v = Start)$
   $\wedge\, (Count_u\{P(v, u)\} = 1 \vee Last(v))$,
6. All vertices are reachable from the starting node: $\forall v : Count_t\{R(v, t)\} = 1 \wedge$
   $\forall t : Count_v\{R(v, t)\} = 1 \wedge (R(Start, 0)) \wedge (\forall u, v, d : R(v, d) \wedge P(v, u) \rightarrow R(u, d + 1))$.

Usually adding extra axioms to a specification makes the output CNF larger but the corresponding extra clauses make the task of SAT solvers easier, by helping SAT solvers to detect inconsistency earlier and not allowing them to spend time in inconsistent branches.

Observation 2 *If vertex v is the d-th vertex in a Hamiltonian path P, and there is no path of length i from v to u in G, then u cannot be the $d + i$-th vertex of path P.*

Observation 3 *The following set of inflationary fixpoint rules defines $Dist(u, v, d)$, such that $Dist($ u, v, $d)$ is true iff there is a path of length d from u to v:*

$$
\begin{aligned}
&\{Dist(u, v, d) \leftarrow Edge(u, v) \wedge d = 1.\}\\
&\{Dist(u, v, d) \leftarrow \exists w, d_0 : (d = 2 \cdot d_0 \wedge\\
&\qquad\qquad Dist(u, w, d_0) \wedge Dist(w, v, d - d_0))\\
&\qquad\qquad \vee\, (d = 2 \cdot d_0 + 1 \wedge\\
&\qquad\qquad Dist(u, w, d_0) \wedge Dist(w, v, d - d_0)).\}
\end{aligned}
$$

Using Observations 2 and 3, we can infer that the following axiom can be added to the proposed specification for the Hamiltonian path problem:

$$\forall v, d : R(v, d) \rightarrow (\forall u, n : \neg Dist(v, u, n) \rightarrow \neg R(u, d + n))\,. \tag{3.1}$$

In Axiom 3.1, we used a ternary predicate to assert a guard on the binary predicate R. We can also use the following observation to construct a more compact upper bound for R.

Observation 4 *If there is no path in graph G with length d which connects vertex $Start$ to vertex v, vertex v cannot be the d-th vertex in any Hamiltonian path, i.e., we can be sure $R(v, d)$ is false.*

The following set of inflationary inductive definition rules defines $UR(v, d)$ such that $UR(v, d)$ is true iff there is a path of length d from $Start$ to v:

$$\{UR(v, d) \leftarrow v = Start \wedge d = 0\}.$$
$$\{UR(v, d) \leftarrow \exists u : UR(u, d - 1) \wedge Edge(u, v)\}.$$

Predicate $UR(.,.)$, defined as above, describes an upper bound for predicate $R(.,.)$, and hence we can use UR to bound R, in our axiomatizations.

In Table 3.4, HP-01 refers to the specification expressing the axioms 1-6, in pure first-order logic. Specification HP-02 has only axioms 1 to 6. Specification HP-03 is constructed from axioms 1-6 plus the axioms defining $Dist(.,.,.)$ predicate and relating $Dist$ and R predicates. The last specification, HP-04, describes predicate UR using the above set of inflationary fixpoint axioms, includes axiom 1 to 5, and has the following rewritten version of axiom 6:

$\forall v : Count_t\{R(v, t) \wedge UR(v, t)\} = 1 \wedge \forall t : Count_v\{R(v, t) \wedge UR(v, t)\} = 1 \wedge (R(Start, 0)) \wedge (\forall u, v, d : R(v, d) \wedge P(v, u) \wedge UR(v, d) \wedge UR(u, d + 1) \rightarrow R(u, d + 1))$.

Specification	Axioms	Number Solved	Grounding Time	Solving Time
HP-01	FO	$56/58$	4.3	14.5
HP-02	Count	$58/58$	0.7	30.6
HP-03	Count + Poor Guards	$2/58$	361.0	4.4
HP-04	Count + Upper Guards	$58/58$	2.8	11.27

Table 3.4: Comparison of the performance of different specifications of the same problem. The best result, shown in bold, is obtained by HP-04.

3.3.5 Summary of Techniques for Developing Efficient Specifications

So far, we have illustrated techniques that can be used to make a given specification more efficient. To apply these techniques, users need to know the domain well and use their knowledge of the problem for rewriting/developing specifications. For example, from our knowledge about the domain of the Blocked N-Queens problem, we observed that $\neg Blk(r, c)$ is an upper bound for $Q(r, c)$. In the Graph Colouring problem, we knew that the search space of this problem has certain kinds of symmetries, and to break the symmetries, we developed appropriate axioms using the features available in the input language of Enfragmo.

From the results of our experiments, it can be concluded that SAT solvers perform better when the search space is smaller. Essentially the aim of all of the techniques introduced in this section, is to reduce the size of search space, without changing the satisfiability of the instance. We reduced

the size of the search space by adding symmetry-breaking axioms and by altering the axioms to use the information about upper and lower bounds.

Crawford showed that the computational complexity of the symmetry detection problem on specifications developed in first-order logic is equivalent to the Graph Isomorphism problem [26]. The graph isomorphism problem is one of the few NP problems which has neither a polynomial time algorithm nor is known to be an NP-complete problem. Since computing full symmetry-breaking axioms is time consuming, different approaches have been developed to partially detect and break the symmetries in the problem search space. One can name [27], [38] and [66] as examples of such approaches.

As we mentioned, the IDP system has a mechanism for automatically inferring upper and lower bounds, and in our group, a similar technique has been developed [69]. As the results of experiments in [69] suggest, enabling this feature adds some overhead to the grounding phase and usually reduces the solving times. Sometimes, the additional time Enfragmo spends in the grounding phase is much bigger than the time it saves in the solving phase. We observe the same pattern in the IDP system, i.e., for some problems, disabling the bounds inference makes the whole solving process faster.

On the other hand, a user who knows the domain well is usually able to detect the important symmetries easily, and may even use some of the available tools to find these symmetries. Then, to break the symmetries, we can augment the problem specification by extra axioms that are satisfied by exactly one of the symmetric solutions [27]. We used the same approach in axiomatizing the Graph Colouring and Social Golfers problems.

The techniques introduced in this chapter can be summarized as follows:
Adding Extra Axioms enables SAT solvers to detect inconsistencies, earlier.
Breaking Symmetries helps the low-level solver to solve ground problems easier/faster.
Using Upper (Lower) Guards in the Specifications helps the grounder to ground faster, produces smaller ground program by removing redundancy (usually, the smaller the ground programs the faster the solving phase).
Using Inductive Definitions allows computing bounds, guards and predicates which can speedup the solving phase.
Using Aggregates enables us to express certain properties more compactly and more intelligibly. For example, compare the following FO axiom expressing the second symmetry breaking axiom of the Social Golfers problem with the original one ($\forall w, g_1, g_2 : Succ(g_1, g_2) \rightarrow Min_p\{p; M(w, g_1, p); 0\} < Min_p\{p; M(w, g_2, p); 0\}$):

vs

$$\begin{aligned}
&\forall w, g_1, g_2 : Succ(g_1, g_2) \rightarrow \forall p_1, p_2(\\
&(M(w, g_1, p_1) \wedge \forall p' : (p' < p_1) \rightarrow \neg M(w, g_1, p')) \wedge && //p_1 \text{ is the smallest indexed player in group } g_1 \\
&(M(w, g_2, p_2) \wedge \forall p' : (p' < p_2) \rightarrow \neg M(w, g_2, p')) && //p_2 \text{ is the smallest indexed player in group } g_2 \\
&) \rightarrow p_1 < p_2.
\end{aligned}$$

In addition to the convenience, we get speed-up when using aggregates: We showed that using Count aggregates to bound the size of sets results in faster solving. We added the FO counterpart

of symmetry breaking axioms to specification SG-01. The new specification failed to solve medium and large size instances.

We believe the axiom used in the Social Golfer problem is easier to read/understand than the above axiom. Therefore having access to aggregates in a language makes the task of expressing complicated properties easier. In addition, using aggregates in the specification decreases the total processing time.

3.4 Experimental Evaluation

In this section, we compare the performance of Enfragmo with that of other grounding-based systems. A set of NP-hard of problems were chosen from the second Answer Set Programming Competition [31], on which the techniques we described can be applied. All the instances and scripts used in this experiment and the other experiments presented in this thesis can be downloaded from [2].

The other solvers used in our experiments are: Clingo (v 3.0.3) [40], DLV (v 2010-10-14) [29], and IDP (v 2.20) [73]. For each system, we used specifications provided by the system's authors, obtained from [31] or [1]. The experiments were run on an Intel Xeon E5530 quad-core 2.4 GHz processor, with a timeout of 1800 seconds.

In this set of experiments, we used the default configurations for all the solvers. The default configuration for Enfragmo is as follow:

1. GroundingMode: TFGrounder
2. SortMode: QuickSort
3. CountMode: DC
4. CNFGenerator: FillAndRewrite
5. CNFGateFlattening: Enabled(1)
6. CNFTseitinMemorization: Enabled(1)

Since the input language accepted by each of the solvers is different, the same specification cannot be used for all the solvers. Since the first-order fragment of the input languages of Enfragmo and IDP are almost the same, we used the same pure first-order specifications for Enfragmo and IDP. It should be mentioned that the syntax of the input languages of Enfragmo and IDP are very similar, but there are differences between the semantics of these two languages. Although both DLV and Clingo are both ASP solvers, their input languages are very different.

The rows in Table 3.5 correspond to problems, and the columns correspond to the fastest specification of each solver. Table 3.5 also has two special columns, Enf-FO and IDP-FO. These two columns show the performance of Enfragmo and IDP on the same first-order specification.

Table 3.5 shows that Enfragmo was able to solve almost all instances in the collection, and performed the best on five of the nine problems. Enfragmo also performed relatively well on the first-order specifications, which a naive user may develop.

3.5 Conclusion

We have presented the Enfragmo system for modelling and solving combinatorial search problems. It provides users with a convenient way to specify and solve computationally hard problems, in particular search problems whose decision versions are in the complexity class NP.

Problem	# of Ins	Clingo	DLV	IDP	Enfragmo	IDP-FO	Enf-FO
GraphCol	105	70/342.3	20/564.8	24/159.6	**105/0.9**	24/159.6	95/360.0
HamPath	58	58/0.21	13/385.4	**58/0.1**	58/17.3	58/9.9	58/14.1
SocGol	175	134/6.34	96/541.3	142/2.7	**162/19.67**	78/46.06	126/53.2
SchrNum	29	29/30.6	18/92.9	28/30.4	**29/22.2**	28/30.4	**29/22.2**
BlQueens	400	400/7.6	**400/0.4**	395/52.2	400/28.6	385/56	264/369.2
ConnDomSet	21	20/15.1	13/362.2	18/128.8	**21/41.4**	14/163.4	**21/41.4**
DisjSched	10	10/117.4	5/358.1	10/78.5	**10/42.1**	5/59.2	**10/42.1**
15 Puzzle	27	**27/24.3**	9/247.6	27/63.6	27/396.8	3/325.0	27/599.0
HierClust	12	**12/0.08**	12/2.7	12/1.9	12/11.7	12/2.4	12/11.7

Table 3.5: Performance comparison of Enfragmo and other systems. The entry n/t means that n instances were solved in total time of t seconds. The best result for each problem is in bold.

In the context of declarative specification-based problem solving, we demonstrated the features of specifications that produce improved system performance over otherwise simpler but equivalent specifications. We observed that, while predicting the effects of a particular change is not necessarily possible, the improvements come not from sheer trial and error, but from applying an understanding of the structure of the domain.

While the results of experimental comparisons are not precise, they confirm that Enfragmo is competitive with existing solvers. Our experiments indicate that Enfragmo is already competitive with other similar systems when using the most straightforward specifications, and its performance can be sped up considerably by applying the described techniques. Our techniques can be used in other systems with the appropriate language constructs. While the techniques are general, the exact nature of the effect on performance will, of course, be system dependent.

Chapter 4

Expressive Power of the Input Language of Enfragmo

In this chapter, we show that every problem expressible in GGF_k logic can be expressed in the input language of Enfragmo, and using this connection, we relate the expressive power of the input language of Enfragmo and GGF_k logic.

4.1 Introduction

There are many logic programming-based systems, each of which accepts/works with its own language. The main goal in developing these languages is to make the task of problem description easier. Sometimes, having certain constructs in a specification language increases the expressive power of the language and allows users to describe properties/problems which cannot be solved by the corresponding solver. The input language of DLV-complex, described in Subsection 7.3.1, is an example of such languages. We believe it is always informative to study the expressive power of logic programming-based solvers' languages. Our aim, in this chapter, is to describe the relation between GGF_k logic [65] and the input language of Enfragmo, and show that the complexity class expressible by the input language of Enfragmo is *NP*. Also, we describe a fragment of first-order logic that corresponds to the input language of Enfragmo.

In [54], Mitchell and Ternovska emphasized the importance of capturing NP for the languages used in specification based systems. According to the authors of [54], the capturing property is a necessity since it shows that for a given language:

- Users can express all problems in NP: Users know that their NP problem can be certainly specified for the system,
- Users can not express more than NP: This feature ensures that solving can be achieved by constructing a universal poly-time reduction to an NP-complete problem such as SAT.

In the previous chapter, we discussed that supporting aggregates and arithmetic operators is a desirable feature for a specification language/solver. In order to both allow arithmetic operators in the input language and stay in NP, we have to use logics that are fragments of first-order logic. As we mentioned in Section 2.5, GGF_k and $DGGF_k$ logics are fragments of first-order logic enriched with

arithmetic and inductive definitions. These logics are designed carefully such that the embedded model expansion on these logics when the background structures is as what defined by Definition 7, captures a subset of problems in NP. Therefore by using a specification in GGF_k ($DGGF_k$) logic we can be certain that a polynomial time reduction to an instance of the SAT problem exists.

The structure of this chapter is as follows. In Section 4.2, we describe how to construct a model preserving translation from formulas in GGF_k logic to specifications for Enfragmo, and hence we show that every NP problem expressible in GGF_k logic is also expressible in the input language of Enfragmo. In Section 4.3, we describe a logic which corresponds to the input language of Enfragmo.

4.1.1 My Contributions

The translation of formulas in GGF_k logic to specifications the input language of Enfragmo is contributed by the author. The two methods for expressing Product operator, explained in Subsections 4.2.2 and 4.2.3, are proposed by the author of this thesis.

4.2 Expressive Power of the Input Language of Enfragmo

In this section, we show that every NP problem satisfying the small-cost condition has a specification in the input language of Enfragmo. To this aim, we describe three constructions which transform every MX specification expressed in $GGF_k(\varepsilon)$, with the background structure as defined in Definition 7, to a specification for Enfragmo. Since we know that every NP problem satisfying small-cost condition can be expressed in $GGF_k(\varepsilon)$, we conclude that all such NP problems can be expressed in the input language of Enfragmo. Our proof is based on the description of algorithms which transform formulas in $GGF_k(\varepsilon)$ with the background structure as defined in Definition 7 to specifications in the input language of Enfragmo. In this chapter, whenever we refer to $GF F_k(\varepsilon)$ logic, we assume the background structure is as what is defined in Definition 7.

In GGF_k, the variables do not have specific sorts, and the possible sets of instantiations of variables are restricted by lower guards. Also, this logic requires expansion predicates to be bounded by upper guards. On the other hand, in the input language of Enfragmo, each variable in the specification has a corresponding sort (type), and every expansion predicate is implicitly guarded. In this section, we explain how to transform formula $\Phi \wedge \Psi$ expressed in $GGF_k(\varepsilon)$ (Definition 6), and structure $\mathcal{A}$, describing the instance structure, to an Enfragmo problem specification, $Spec(\Phi \wedge \Psi)$, and an Enfragmo problem instance, $Ins(\Phi \wedge \Psi, \mathcal{A})$. In this chapter, we assume structure $\mathcal{A}$ satisfies the small-cost condition, i.e., the value of the maximum integer in the active domain of $\mathcal{A}$ is less than $2^{poly(|adom_{\mathcal{A}}|)}$, where $poly$ is a polynomial.

Given formula $\Phi \wedge \Psi$ in $GGF_k(\varepsilon)$, we extract the set of relations in the expansion vocabulary, ε, from the upper guard axioms in Ψ, and set the instance vocabulary to be $\sigma = vocab(\Phi \wedge \Psi) \backslash \varepsilon$.

Let $\Phi \wedge \Psi$ be a formula in $GGF_k(\varepsilon)$ and ϕ be a subformula of Φ. We construct specification $Spec(\Phi \wedge \Psi)$ in the input language of Enfragmo, using the following two methods:

- Method $FormulaTrans(\phi)$ translates $\phi \in GGF_k(\varepsilon)$ to a formula in the input language of Enfragmo;

- Method $TermTrans(t)$ translates term t in $GGF_k(\varepsilon)$ to a term in the input language of Enfragmo (We explain how $TermTrans(t)$ works later).

1. Introduce "ADOMType" as the only Type in $Spec(\Phi \wedge \Psi)$;
2. Define specification $Spec(\Phi \wedge \Psi)$ to have exactly one phase;
3. Set the axiom in the only phase of $Spec(\Phi \wedge \Psi)$ to be $FormulaTrans(\Phi)$;
4. If ϕ is in form $\phi_1 \wedge \phi_2$, then set $FormulaTrans(\phi)$ to be $FormulaTrans(\phi_1) \wedge FormulaTrans(\phi_2)$;
5. If ϕ is in form $\phi_1 \vee \phi_2$, then set $FormulaTrans(\phi)$ to be $FormulaTrans(\phi_1) \vee FormulaTrans(\phi_2)$;
6. If ϕ is in form $\neg\phi_1$, then set $FormulaTrans(\phi)$ to be $\neg FormulaTrans(\phi_1)$;
7. If ϕ is in form $\exists\bar{x}(G_1 \wedge \cdots G_m \wedge \psi)$, where G_i are atomic formulas, then set $FormulaTrans(\phi)$ to be $\exists\bar{x}(G_1 \wedge \cdots G_m \wedge FormulaTrans(\psi))$, and in $Spec(\Phi \wedge \Psi)$, set the sort of variables in $\bar{x}$ to be ADOMType;
8. If ϕ is in form $P(t_1(\bar{x}), \cdots, t_m(\bar{x}))$, where P is a predicate, then,
   - Define predicate P, in specification $Spec(\Phi \wedge \Psi)$, to be an m-ary predicate, if it is the first time we visit P,
   - Associate each argument of predicate P, in specification $Spec(\Phi \wedge \Psi)$, with ADOMType;
   - Set $FormulaTrans(\phi)$ to be $P(T_1(\bar{x}), \cdots, T_m(\bar{x}))$, where each T_i is the transformed form of t_i, computed using $TermTrans(t_i)$.
9. If ϕ is in form $t_1(\bar{x})\ op\ t_2(\bar{x})$, where *op* is a comparison operator, then set $FormulaTrans(\phi)$ to be $T_1(\bar{x})\ op\ T_2(\bar{x})$, where T_i, $i = 1, 2$, is the transformed form of t_i, computed using $TermTrans(t_i)$;
10. If ψ is an upper guard axiom in Ψ, such that ψ is in form $\forall\bar{x}\ (E(\bar{x}) \rightarrow G_1(\bar{x}) \wedge \cdots G_m(\bar{x}))$:
    - In $Spec(\Phi \wedge \Psi)$, define E as an expansion predicate and also define each of its arguments to be from type ADOMType;
    - Add axiom $\forall\bar{x}\ E(\bar{x}) \rightarrow G_1(\bar{x}) \wedge \cdots G_m(\bar{x})$ to $Spec(\Phi \wedge \Psi)$, where the type of each variable in $\bar{x}$ is ADOMType.

Before describing how we can construct $TermTrans$, we briefly discuss how to construct an Enfragmo instance from instance structure $\mathcal{A}$. Transforming instance structure $\mathcal{A}$ to Enfragmo problem instance $Ins(\Phi \wedge \Psi, \mathcal{A})$ is straightforward: For each predicate, I, which is not defined an expansion predicate in $Trans(\Phi \wedge \Psi)$, add tuple $\bar{a}$ to the interpretation of I in $Ins(\Phi \wedge \Psi, \mathcal{A})$, iff we have $\bar{a} \in I^{\mathcal{A}}$. We also set $ADOMType$ in $Ins(\Phi \wedge \Psi, \mathcal{A})$ to be the active domain of $\mathcal{A}$.

Now, we define the transformation for terms, $TermTrans(t)$, recursively as follows:

1. If t is a variable x, then set $TermTrans(t)$ to be x.
2. If t is a constant c, then introduce a new instance function, f_c, in problem specification $Spec(\Phi \wedge \Psi)$. In problem instance $Ins(\Phi \wedge \Psi, \mathcal{A})$, set the interpretation of f_c to c. Finally, set $TermTrans(t)$ to f_c.
3. If t is in form $f(t_1, \cdots, t_n)$, where f is an n-arity function, then define f as an instance function in specification $Spec(\Phi \wedge \Psi)$. Set $TermTrans(t)$ to $f(TermTrans(t_1), \cdots, TermTrans(t_n))$.

4. If t is in form t_1 *op* t_2 where *op* is an arithmetical operator, then set $TermTrans(t)$ to $TermTrans(t_1)$ *op* $TermTrans(t_2)$.
5. If t is in form $\chi(\phi(\bar{x}))$, then set $TermTrans(t)$ to $Max_{\emptyset}\{1 : \phi(\bar{x}); 0\}$.
6. If t is in form $min_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, then set $TermTrans(t)$ to $Min_{\bar{x}}\{TermTrans(f) : FormulaTrans(\phi); 0\}$ where the types of all variables in $\bar{x}$ are ADOMType.
7. If t is in form $max_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, then set $TermTrans(t)$ to $Max_{\bar{x}}\{TermTrans(f) : FormulaTrans(\phi); 0\}$ where the types of all variables in $\bar{x}$ are ADOMType.
8. If t is in form $\sum_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, then set $TermTrans(t)$ to $Sum_{\bar{x}}\{TermTrans(f) : FormulaTrans(\phi)\}$ where the types of all variables in $\bar{x}$ are ADOMType.
9. If t is in form $\prod_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, since Enfragmo does not have a multiset operator (aggregate operator) corresponding to $\prod$, we need to express $\prod$ using the other constructs available in the language of Enfragmo. In the rest of this section, we describe how this can be achieved.

Proposition 1 *Given formula $\Phi \wedge \Psi \in GGF_k(\varepsilon)$ and structure $\mathcal{A}$, such that $\prod$ does not occur in $vocab(\Phi \wedge \Psi)$, the above construction generates specification $S = Tran(\Phi \wedge \Psi)$ and instance $I = Ins(\Phi \wedge \Psi, \mathcal{A})$ such that there is a structure expanding $\mathcal{A}$ and satisfying $\Phi \wedge \Psi$ iff Enfragmo finds a solution for specification S and instance I. Also, every structure $\mathcal{B}$ expanding $\mathcal{A}$ and satisfying $\Phi \wedge \Psi$ can be translated to a solution for specification S and instance I, and vice versa. Specification S does not depend on the instance structure and is computed solely based on $\Phi \wedge \Psi$. The set of integers in the active domain of $\mathcal{A}$ is the same as the set of integers in $Ins(\Phi \wedge \Psi, \mathcal{A})$.*

The proof of the above proposition is mechanical and involves using structural induction on the formula tree of $\Phi \wedge \Psi$. The last part of the proposition can be verified by looking at the procedure described for constructing $FormulaTrans$, $TermTrans$ and Ins. Since none of these procedures introduces a new element, the set of integers in the active domain of $\mathcal{A}$ is the same as the set of integers in the instance file constructed for $\mathcal{A}$.

The proof of capturing result for $GGF_k(\varepsilon)$, presented in [65], does not work if we remove $\prod$ multiset term from this logic. So, to show the capturing result for Enfragmo, we have to find a way(s) to express terms involving the $prod$ multiset operator in an Enfragmo specification. The author of this thesis proposed the following three approaches to construct specifications from formulas in GGF_k which use $\prod$:

1. Adding *Product* aggregate to the input language of Enfragmo,
2. Expressing $\prod$ using Max aggregate,
3. Expressing $\prod$ using Sum aggregate.

Before describing how to express $\prod$ term in the input language of Enfragmo, we use an example to illustrate how the above translation works.

Example 6 *(Continuation of Example 3) Here we translate the formula in GGF_k logic, presented in Example 3, to a specification in Enfragmo.*

Recall that in Example 3, Φ and Ψ are $(\Sigma_x(w(x) : O(x) \wedge O'(x)) \leq b) \wedge (k \leq \Sigma_x(v(x) : O(x) \wedge O'(x)))$, and $\forall x : O'(x) \rightarrow O(x)$, respectively.

Method $Spec(\Phi \wedge \Psi)$ generates the following specification:

GIVEN:
TYPES: ADOMType;
PREDICATES: O(*ADOMType*), O'(*ADOMType*);
FUNCTIONS: k :*ADOMType*, b :*ADOMType;*
FIND: O'*;*
PHASE:
SATISFYING:
//FormulaTrans(Φ*)*
$(Sum_x\{w(x) : O(x) \wedge O'(x)\} \leq b) \wedge (k \leq Sum_x\{v(x) : O(x) \wedge O'(x)\})$;
PRINT: O, O';

In the rest of this section, we describe how each of the proposed approaches deals with the $\prod$ multiset operator.

4.2.1 Expressing $\prod$ using Product Aggregate

The input language of Enfragmo does not have the Product aggregate. We can add this aggregate to the input language of Enfragmo, and modify Enfragmo's grounding algorithm to work on the new language. Assuming Enfragmo supports the Product aggregate, we set the $TermTrans(\prod_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\})$ to be:

$$Product_{\bar{x}}\{TermTrans(f) : FormulaTrans(\phi)\},$$

where the corresponding type of each variable in $\bar{x}$ is ADOMType.

So, by adding Product aggregate to the input language of Enfragmo, and extending the grounding algorithm, we can construct specification S and instance I from $\Phi \wedge \Psi \in GGF_k(\varepsilon)$ and structures $\mathcal{A}$ such that specification S has a constant size (assuming formula $\Phi \wedge \Psi$ is fixed) and instance I has polynomial size with respect to the size of representation of $\mathcal{A}$. The set of elements in the active domain of $\mathcal{A}$ is the same as the set of values in I.

4.2.2 Expressing Product using Max Aggregate

Given term $t(\bar{y}) = \prod_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, and having access to the instance structure $\mathcal{A}$, we can express $t(\bar{y})$ using Max aggregate. Let $\bar{a}_1, \cdots, \bar{a}_n$ be the tuples in $adom_{\mathcal{A}}^m$ where $m = |\bar{x}|$ satisfying the lower guard on $\bar{x}$ (Recall that we use $adom_{\mathcal{A}}^m$ to denote set $\{\langle c_1, \cdots, c_m\rangle \mid c_i \in adom_{\mathcal{A}}, i = 1 \cdots, m\}$). So, the result of the following multiplication is equal to $t(\bar{y})$:

$$\begin{aligned}
&(Max_{\emptyset}\{f(\bar{a}_1,\bar{y}) : \phi(\bar{a}_1,\bar{y}) : 1\}) \times \\
&(Max_{\emptyset}\{f(\bar{a}_2,\bar{y}) : \phi(\bar{a}_2,\bar{y}) : 1\}) \times \\
&\qquad\qquad\qquad\qquad\qquad\qquad\vdots \\
&(Max_{\emptyset}\{f(\bar{a}_n,\bar{y}) : \phi(\bar{a}_n,\bar{y}) : 1\}) .
\end{aligned}$$

In the above equation, each Max aggregate returns, for different assignment to $\bar{y}$, either $f(\bar{a}_i,\bar{y})$ or 1 depending on the truth value of $\phi(\bar{a}_i,\bar{y})$. The result of the product of the above n Max aggregates is the same as the result of the corresponding Product term.

In order to define $TermTrans(t(\bar{y}))$ as above, we need to introduce n fresh constant symbols, $g_1, \cdots, g_n$, in $FormulaTrans(\Phi \wedge \Psi)$, corresponding to $\bar{a}_1, \cdots, \bar{a}_n$, and set the interpretation for these constants appropriately, in $Ins(\Psi \wedge \Psi, \mathcal{A})$, and set the $TermTrans(t(\bar{y}))$ be:

$$\begin{aligned}(Max_{\emptyset}\{f(g_1, \bar{y}) : \phi(g_1, \bar{y}) : 1\}) \times \\ (Max_{\emptyset}\{f(g_2, \bar{y}) : \phi(g_2, \bar{y}) : 1\}) \times \\ \vdots \\ (Max_{\emptyset}\{f(g_n, \bar{y}) : \phi(g_n, \bar{y}) : 1\}) .\end{aligned}$$

This way of translating $\prod$ creates specifications whose sizes are a function of the size of $adom_{\mathcal{A}}$. However, the set of elements in the active domain of $\mathcal{A}$ is the same as the set of values in $Ins(\Phi \wedge \Psi, \mathcal{A})$.

4.2.3 Expressing Product using Sum Aggregate

In order to use the transformation presented in the previous Subsection to generate a problem specification, we need to have access to the instance structure. So, we cannot claim that every problem expressible in GGF_k has a corresponding specification in Enfragmo. The other issue with the previous construction is that the size of the resulting specification is not constant (the size depends on the size of $\mathcal{A}$). Here, we describe another approach for constructing specifications for Enfragmo solely from formulas in GGF_k. Therefore, we can have a constant size specification for every problem expressible in GGF_k logic. Without loss of generality, we assume the range of function f, used in $\prod$ multiset operator, is positive integers. With a slight modification, this approach can be extended to the general case.

Let $t(\bar{y}) = \prod_{\bar{x}}\{f(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y})\}$ be a term of $\Phi \wedge \Psi \in GGF_k$ and $t_{\bar{a}}(\bar{y})$ be the following term:

$$t_{\bar{a}}(\bar{y}) = \prod_{\bar{x}}\{f(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}) \wedge (\bar{x} \leq \bar{a})\}.$$

The following recurrence relation defines the value of $t_{\bar{a}}(\bar{y})$:

$$t_{\bar{a}}(\bar{y}) = \begin{cases} 1 & \bar{a} \text{ is the minimum value in } \bar{x} \text{ may take and } \phi(\bar{a}, \bar{y}) \text{ is } \textit{false}; \\ f(\bar{a}, \bar{y}) & \bar{a} \text{ is the minimum value } \bar{x} \text{ may take and } \phi(\bar{a}, \bar{y}) \text{ is } \textit{true}; \\ t_{\bar{b}}(\bar{y}) & \bar{b} \text{ is the predecessor of } \bar{a} \text{ and } \phi(\bar{a}, \bar{y}) \text{ is } \textit{false}; \\ t_{\bar{b}}(\bar{y}) \times f(\bar{a}, \bar{y}) & \bar{b} \text{ is the predecessor of } \bar{a} \text{ and } \phi(\bar{a}, \bar{y}) \text{ is } \textit{true}. \end{cases}$$

Let $\bar{M}$ denote the maximum value $\bar{x}$ may take, based on the lower guards for $\bar{x}$. We have $t(\bar{y})$ and $t_M(\bar{y})$ evaluate to the same value, in all structures $\mathcal{B}$ expanding $\mathcal{A}$.

In the following, we describe a transformation which encodes the above recurrence relation. In order to define the value of term $t_{\bar{a}}(\bar{y})$, we extend the vocabulary of and add some extra axioms to the specification. More formally, for term $t(\bar{y})$, we modify the specification and instance as follows (recall that $\bar{x}$ is the tuple of quantified variables in term $t(\bar{y}) = \prod_{\bar{x}}\{f(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y})\}$):

1. Introduce two new integer types Range$_t$ and Power2$_t$, in the specification. Let us call the maximum value term $t(\bar{y})$ that can be obtained, under all possible assignments to $\bar{y}$ and models expanding $\mathcal{A}$, MAX. In $Ins(\Phi \wedge \Psi, \mathcal{A})$, we define the interpretation for Range$_t$, $Range_t$, to be the set of all integers in range 1 to $\log(MAX)$, inclusive. We also define the interpretation for Power2$_t$, $Power2_t$, to be the set $\{2^r : r \in Range_t\}$;
2. Introduce a new instance function P2$_t(v)$ in the specification, such that P2$_t$ maps values from type Range$_t$ to type Power2$_t$. In $Ins(\Phi \wedge \Psi, \mathcal{A})$, we define $P2_t$ such that it maps v to 2^v, for all v in $Range_t$;
3. Expand the vocabulary of the specification by a new instance function MIN$_t$, where MIN$_t$ is a function without any argument, i.e., it is a constant. In $Ins(\Phi \wedge \Psi, \mathcal{A})$, we define the interpretation of MIN$_t$ such that it returns the minimum value of $ADOMType^m$, where $m = |\bar{x}|$;
4. Expand the vocabulary of the specification by a new instance function MAX$_t$, where MAX$_t$ is a function without any argument. In $Ins(\Phi \wedge \Psi, \mathcal{A})$, we define the interpretation of MAX$_t$ such that it returns the maximum value of $ADOMType^m$, where $m = |\bar{x}|$. Note that MAX$_t$ is an instance function while MAX, used in (1), is an integer;
5. Introduce a new instance predicate Succ$_t(\cdots)$, such that it has $2|\bar{x}|$ arguments and all its arguments are from type ADOMType. In $Ins(\Phi \wedge \Psi, \mathcal{A})$, we define Succ$_t$ such that Succ$_t(\bar{a}, \bar{b})$ is *true* iff $\bar{b}$ is the successor of $\bar{a}$, where $\bar{a}, \bar{b} \in ADOMType^m$, where $m = |\bar{x}|$;
6. Introduce a new expansion predicate E$_t(\cdots)$, with $2|\bar{x}| + 1$ arguments , such that the first $2|\bar{x}|$ arguments are from type ADOMType, and the last argument is from type Range$_t$.

We add appropriate axioms to define $E_t(\bar{z}, \bar{w}, v)$, where both $\bar{z}$ and $\bar{w}$ are tuples of size m of variables from type ADOMType, and v is a variable from type Range$_t$, such that for $\bar{a} \in ADOMType^m$, $v \in Range_t$ and for every instantiation γ, mapping variables in $\bar{w}$ to elements in $ADOMType$, we have $E_t(\bar{a}, \gamma|_{\bar{w}}, v)$ is ***true*** iff:

The v-th bit, in the binary representation, of the result of evaluation of term $t_{\bar{a}}(\gamma|_{\bar{w}})$, is one.

If we define predicate E_t as above, the value of $t_{\bar{a}}(\bar{y})$ is equal to the result of the following Sum aggregate:

$$Sum_v\{P2_t(v) : E_t(\bar{a}, \bar{y}, v)\}.$$

The following axioms define E_t (Recall that ϕ is the formula used as the condition in the product multiset operator):

1. $\forall \bar{y}\ \phi(MIN_t, \bar{y}) \rightarrow Sum_v\{P2_t(v) : E_t(MIN_t, \bar{y}, v)\} = f(MIN_t, \bar{y})$;
2. $\forall \bar{y}\ \neg\phi(MIN_t, \bar{y}) \rightarrow Sum_v\{P2_t(v) : E_t(MIN_t, \bar{y}, v)\} = 1$;
3. $\forall \bar{y}, \bar{x}, \bar{x}'\ (\neg\phi(\bar{x}', \bar{y}) \wedge Succ_t(\bar{x}', \bar{x})) \rightarrow (\forall v\ E_t(\bar{x}', \bar{y}, v) \leftrightarrow E_t(\bar{x}, \bar{y}, v))$;
4. $\forall \bar{y}, \bar{x}, \bar{x}'\ (\phi(\bar{x}', \bar{y}) \wedge Succ_t(\bar{x}', \bar{x})) \rightarrow Sum_v\{P2_t(v) : E_t(\bar{x}', \bar{y}, v)\} = f(\bar{x}', \bar{y}) \times Sum_v\{P2_t(v) : E_t(\bar{x}, \bar{y}, v)\}$.

Finally, we set $TermTrans(t(\bar{y}))$ to be the following:

$$Sum_v\{P2_t(v) : E_t(MAX_t, \bar{y}, v)\}.$$

The following lemmas and propositions assert the properties of the above transformation.

Lemma 1 *Given instance structure $\mathcal{A}$, and formula $\Phi \wedge \Psi$ in GGF_k logic, both the maximum and the minimum values a term in $\Phi \wedge \Psi$ can take, on all possible assignments and all structures $\mathcal{B}$ expanding $\mathcal{A}$, are bounded by $2^{poly(|adom_{\mathcal{A}}|)}$, where $poly(.)$ is a polynomial. There is also a polynomial time algorithm, with respect to $|adom_{\mathcal{A}}|$, to compute an upper bound (a lower bound) for the maximum (minimum, respectively) value term t can take, under all assignments and all models expanding $\mathcal{A}$, assuming representation of term $\Phi \wedge \Psi$ has a constant size.*

Proof: We prove the first part of the lemma by induction on the structure of term t.

The base cases are when t is a variable, an instance of χ function, characteristic function, or a function without any argument. We show the proof for two of base cases. The proof for the other case is similar.

- Term t is $\chi[\phi(\bar{x})]$: The maximum (minimum) value term t can take is 1 (0, respectively), which is clearly bounded by $2^{poly(|adom_{\mathcal{A}}|)}$.
- Term t is a variable: Since all variables are guarded by instance predicates, the maximum (minimum) value a variable can take is the largest (smallest, respectively) integer in the instance structure. Since $\mathcal{A}$ satisfies small-cost property, the maximum (minimum) value in the active domain of $\mathcal{A}$ is bounded by $2^{poly(|adom_{\mathcal{A}}|)}$.

Here we just show the inductive step for terms in form $t(\bar{y}) = \prod_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$. The proofs for the other cases are similar.

By inductive hypothesis, the maximum value $f(\bar{x},\bar{y})$ can take is bounded by two to the power of a polynomial, $poly_1$. Let $G(\bar{x})$ be the conjunction of lower guards on $\bar{x}$ and n_G be the number of tuples satisfying $G(\bar{x})$. Since $G(\bar{x})$ is composed of instance predicates, we can compute the exact value of n_G, efficiently. We have:

$$\begin{aligned}\prod_{\bar{x}}\{f(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\} \leq \prod_{\bar{x}}\{f(\bar{x},\bar{y}) : G(\bar{x}) \wedge \top\} \leq \left(2^{poly_1(|adom_{\mathcal{A}}|)}\right)^{n_G} \\ \leq \left(2^{poly_1(|adom_{\mathcal{A}}|)}\right)^{|adom_{\mathcal{A}}|^{|\bar{x}|}} \leq \left(2^{poly_1(|adom_{\mathcal{A}}|)|adom_{\mathcal{A}}|^{|\bar{x}|}}\right), \qquad (4.1)\end{aligned}$$

which is bounded by $2^{poly_2(|adom_{\mathcal{A}}|)}$ if we select $poly_2(n) = poly_1(n) \times n^{|\bar{x}|}$.

Given structure $\mathcal{A}$, to estimate the maximum value term t can take, we use the following recurrence relation:

$$U_t = \begin{cases} \text{the largest integer in the active domain} & \text{if } t \text{ is either a variable or constant;} \\ \text{the largest integer the in active domain} & \text{if } t \text{ is in the form } f(t_1,\cdots,t_m); \\ U_{t_1} + U_{t_2} & \text{if } t \text{ is in the form } t_1 + t_2; \\ U_{t_1} \times U_{t_2} & \text{if } t \text{ is in the form } t_1 \times t_2; \\ 1 & \text{if } t \text{ is in the form } \chi(\phi(\bar{x})[\gamma]; \\ U_{t_1} & \text{if } t \text{ is in the form } min_{\bar{x}}\{t_1 : \phi\}; \\ U_{t_1} & \text{if } t \text{ is in the form } max_{\bar{x}}\{t_1 : \phi\}; \\ U_{t_1} \times |adom_{\mathcal{A}}|^{|\bar{x}|} & \text{if } t \text{ is in the form } \sum_{\bar{x}}\{t_1 : \phi\}; \\ U_{t_1}^{|adom_{\mathcal{A}}|^{|\bar{x}|}} & \text{if } t \text{ is in the form } \prod_{\bar{x}}\{t_1 : \phi\}. \end{cases} \qquad (4.2)$$

Using structural induction on the parse tree of term t, similar to the proof of the first part, we can show that the value computed by U_t for any term t in GGF_k, is bounded by $2^{poly(|adom_{\mathcal{A}}|)}$. The value of U_t can be computed using the above recurrence relation in linear time with respect to the size of representation of term t and the size of the active domain of $\mathcal{A}$.

Using a similar idea, one can define recurrence relation L_t to compute a lower bound for the values term t may take. ■

Proposition 2 *If we use the value computed by U_t, described by Equation 4.2, as the value of MAX in our construction, both the size and the maximum value in $Range_t$ are polynomial with respect to the number of elements in the active domain of $\mathcal{A}$.*

Proof: In Proposition 1, we showed that U_t is bounded from above by $2^{poly(|adom_{\mathcal{A}}|)}$, where $poly$ is a polynomial. Therefore, by selecting MAX to be U_t, the number of elements in $Range_t$ would be $\log(MAX) = poly(|adom_{\mathcal{A}}|)$. Set $Range_t$ can be encoded using $O(poly(|adom_{\mathcal{A}}|) \log poly(|adom_{\mathcal{A}}|))$ bits in an Enfragmo problem instance, since we have $poly(|adom_{\mathcal{A}}|)$ integers, each of which can be represented using $\log poly(|adom_{\mathcal{A}}|)$ bits. ■

Proposition 3 *The number of elements in $Power2_t$, is polynomial with respect to the number of elements in the active domain of $\mathcal{A}$. In addition, the size of representation of this set is also polynomial with respect to the number of elements in the active domain of $\mathcal{A}$.*

Proof: The proof of this proposition is very similar to that of Proposition 2. ■

Lemma 1 and Propositions 2 and 3 imply that the size of representation of $Ins(\Phi \wedge \Psi, \mathcal{A})$ is polynomial with respect to the size of the active domain of $\mathcal{A}$, given that $\mathcal{A}$ satisfies small-cost condition.

Proposition 4 *Let $\Phi \wedge \Psi \in GGF_k(\varepsilon)$ and $\mathcal{A}$ be an instance structure satisfying small-cost condition. Using the above construction for transforming a $\prod$ term to a term in the input language of Enfragmo results in an Enfragmo problem specification having constant size and an Enfragmo problem instance having polynomial size with respect to the size $adom_{\mathcal{A}}$.*

Proof: Equipping $TermTrans$ with the Sum aggregate based transformation for $\prod$, we obtain a construction for transforming terms in $GGF_k(\varepsilon)$ to terms in the language of Enfragmo. This construction introduces a constant number of predicates, functions, and axioms to $Spec(\Phi \wedge \Psi)$. Moreover, based on the above lemma and propositions, the size of the resulting instance is polynomial with respect to the size of $adom_{\mathcal{A}}$. ■

We say class $\mathcal{K}$ of arithmetical structures satisfies small cost condition iff all arithmetical structures in $\mathcal{K}$ satisfy small cost condition. Theorem 2 summarizes this section.

Theorem 2 *Every problem satisfying small-cost condition is expressible in the input language of Enfragmo. Also, every problem expressible in the input language of Enfragmo is in NP.*

Proof: To prove this theorem, we first show that every problem satisfying small-cost condition has a specification in Enfragmo and then we show that every specification expressible in the input language of Enfragmo corresponds to a problem in NP, satisfying small-cost condition.

The first part of this theorem is already proven in Proposition 4: Since embedded model expansion on $GGF_k(\varepsilon)$ logic with the arithmetic background structure, as defined in Definition 7, captures all problems in NP, satisfying small-cost condition, using Propositions 4, we conclude that every NP problem, satisfying small-cost condition, can be expressed in the input language of Enfragmo.

As described in Chapter 2, we assume all formulas (specifications) are fixed and instances are finite structures. As we will explain in the next chapters, Enfragmo, given a problem, generates a polynomial size equi-satisfiable SAT instance for any phase of the problem specification, in polynomial time, with respect to the size of the instance structure. Satisfiability of a polynomial size SAT instance is an NP-Complete problem, and there are a constant number of phases in a problem specification. So all problems expressible in the input language of Enfragmo are reducible to SAT, and they belong to the complexity class NP. ■

The rest of this section explains why small-cost constraint is necessary. In [64], Tasharrofi and Ternovska defined language $\mathcal{L}$, which corresponds to the input language of a model expansion based solver, to be active-domain-restricted language iff for all specifications expressible in $\mathcal{L}$, and for all arithmetical structures $\mathcal{A}$ over σ (instance vocabulary) and all expansions of $\mathcal{A}$ such as $\mathcal{B}$ satisfying the specification, we have $adom_{\mathcal{A}} = adom_{\mathcal{B}}$.

In Enfragmo, the domains of all expansion predicates are specified as the cross product of types, which can be thought of as unary instance predicates. So the input language of Enfragmo is active-domain-restricted.

The Integer factorization problem is one of the NP problems that does not satisfy small-cost condition: The instance vocabulary for this problem contains a single constant integer n, and the expansion vocabulary contains two constants, a and b. Given an instance structure $\mathcal{A}$, we are looking for a non-trivial factorization of n, i.e., $n = a \times b$ where $a, b > 1$. Remember that the size of the active domain of the instance structure is one while the value of largest integer in the active domain is n.

Tasharrofi and Ternovska also showed that the integer factorization problem cannot be expressed in any active-domain-restricted language [64]. To express this problem, we need to expand the active domain with some new elements. Since the input language of Enfragmo is an active-domain-restricted language, we cannot express integer factorization for Enfragmo without changing the instance structure. In [64], a logic which captures NP in the presence of built-in arithmetic was proposed.

One way of expressing this integer factorization for Enfragmo is to construct a problem instance with a type of size $\theta(\log n)$, and develop a specification which describes integer factorization based on the binary representation of numbers.

4.3 Corresponding Logic for the Input Language of Enfragmo

As we explained in the previous chapter, a specification for Enfragmo is made up of one or more phases. Each phase consists of two parts; a set of inductive definitions and a set of axioms. The goal of this section is to define the logic corresponding to the input language of Enfragmo.

We define logic GL with respect to a set of unary predicates $G = \{G_1, \cdots, G_m\}$, $GL(G)$, to be the following fragment of FO:

1. It contains all atomic formulas;

2. It is closed under Boolean operations;
3. It contains $\exists x(P(x) \wedge \phi)$, provided $P \in G$ and $\phi \in GL(G)$;
4. It contains $\forall x(P(x) \rightarrow \phi)$, provided $P \in G$ and $\phi \in GL(G)$.

Motivated by GGF_k logic, we define $GGL(G, \varepsilon)$, where $G \cap \varepsilon = \emptyset$, as the set of formulas in the form $\Phi \wedge \Psi$ with $\varepsilon \subseteq vocab(\Phi)$, where Φ is a formula of $GL(G)$ and Ψ is a conjunction of guard axioms, one for each symbol $E \in \varepsilon$ in the form: $\forall \bar{x}\ E(\bar{x}) \rightarrow (P_1(x_1) \wedge \cdots \wedge P_{|\bar{x}|}(x_{|\bar{x}|})$, where $P_i \in G$.

Notice that although $GGL(G, \varepsilon)$ looks similar to $GGF_1(\varepsilon)$, there are some differences between these two logics:

- All the upper guards in $GGL(G, \varepsilon)$ are unary instance predicates from set G, while in $GGF_1($ $\varepsilon)$, there is no restriction on the arity of upper guards.
- In $GGL(G, \varepsilon)$ logic, every variable must occur in exactly one upper guard, while $GGF_1(\varepsilon)$ requires each of the variables to occur in at least one upper guard.

Let E be a set of predicates. We define a $[G, E]$ inductive definition to be a set of $[G, X]$ inductive rules ($X \in E$), such that each $[G, X]$ inductive rule is a formula in the form $\forall \bar{x} : X(\bar{x}) \leftarrow P_1(x_1) \wedge \cdots \wedge P_{|\bar{x}|}(x_{|\bar{x}|}) \wedge \phi(\bar{x})$ where $\phi(\bar{x}) \in GL(G)$ and $P_i \in G$. Symbol X in a $[G, X]$ inductive rule is called the head and the formula on the right hand side of arrow ($\leftarrow$) is the body of the rule. Defined symbols are those that appear in the head of at least one of the rules, and the rest are open symbols. We assume that rules contain no free variables, and every variable that occurs in the head also appears in the body. We use $\Psi\{[G, E]\}$ to denote a $[G, E]$ inductive definition.

A $[G, \varepsilon]$ axiom is a sentence (formula without any free variable) ϕ where $\phi \in GGL(G, \varepsilon)$. A $[G, \varepsilon]$ set of axioms is a collection of $[G, \varepsilon]$ axioms. We use $\Phi\{[G, \varepsilon]\}$ to denote a $[G, \varepsilon]$ set of axioms.

To complete the definition of the syntax of logic, we need to define well-formed terms. We define the well-formed terms in a way similar to Definition 8.

Let G be a set of unary predicates, $\varepsilon_1, \cdots, \varepsilon_k$ be k disjoint sets of predicate symbols and also let $E_1, \cdots, E_k$ be k disjoint sets of predicate symbols such that $E_i \cap \varepsilon_j = \emptyset$, for all $1 \leq i, j \leq k$. Logic $Enf(G, \langle \varepsilon_1, \cdots, \varepsilon_k \rangle, \langle E_1, \cdots, E_k \rangle)$ consists of conjunction of $(\Psi_1\{[G, E_1]\} \wedge \Phi_1\{[G, \varepsilon_1]\}) \wedge \cdots \wedge (\Psi_k\{[G, E_k]\} \wedge \Phi_k\{[G, \varepsilon_k]\})$ where for all $i \leq k$, we have:

1. $vocab(\Psi_i) \backslash E_i \subseteq (vocab(\Psi_1) \backslash E_1) \cup \bigcup_{j=1}^{i-1}(E_j \cup \varepsilon_j)$.
2. $vocab(\Phi_i) \backslash \varepsilon_i \subseteq vocab(\Psi_1) \cup \bigcup_{j=1}^{i} E_j \cup \bigcup_{j=1}^{i-1} \varepsilon_j$;

Now we describe the semantics of logic $Enf(G, \langle \varepsilon_1, \cdots, \varepsilon_k \rangle, \langle E_1, \cdots, E_k \rangle)$.

Given an instance structure $\mathcal{A}$, and formula $\Phi = (\Psi_1\{[G, E_1]\} \wedge \Phi_1\{[G, \varepsilon_1]\}) \wedge \cdots \wedge (\Psi_k$ $\{[G, E_k]\} \wedge \Phi_k\{[G, \varepsilon_k]\}) \in Enf(G, \langle \varepsilon_1, \cdots, \varepsilon_k \rangle, \langle E_1, \cdots, E_k \rangle)$, we say Φ is satisfiable if there is structure $\mathcal{B}$ expanding $\mathcal{A}$ such that $\mathcal{B} \models \Phi$.

As explained in the previous chapter, Enfragmo solves each phase once, independently, without considering the next phases, while a solver for logic Enf must look for a structure satisfying all phases. So the semantics of logic Enf is slightly different from that of the input language of Enfragmo. In some cases, there are models for formulas in logic $Enf(G, \langle \varepsilon_1, \cdots, \varepsilon_k \rangle, \langle E_1, \cdots, E_k \rangle)$ that may not be found by Enfragmo. The following example explains the difference between the two semantics.

Example 7 *Consider problem specification S, in which $E_1(.)$ and $E_2(.)$ are unary expansion predicates, and all variables in the specification are associated with type T. Also, let the interpretation for type T be $\{1, 2\}$. Assume S has two phases:*

1. *The first phase of S has a single axiom: $\exists x\ E_1(x)$.*
2. *The second phase of S also has a single axiom: $\forall x\ E_1(x) \wedge E_2(x)$.*

Based on the semantic of logic Enf, this problem has a single model, in which we have the following facts: $E_1(1), E_1(2), E_2(1)$ and $E_2(2)$.

Now, let us see how Enfragmo acts on this problem: Enfragmo solves the first phase, first. Depending on the answer it receives from the SAT solver, the interpretation for E_1 can be any of the following:

- *Interpretation for E_1 is $\{1\}$: Having this interpretation for E_1, the second phase is unsatisfiable.*
- *Interpretation for E_1 is $\{2\}$: Having this interpretation for E_1, the second phase is unsatisfiable.*
- *Interpretation for E_1 is $\{1, 2\}$: This interpretation for E_1 makes the second phase satisfiable.*

One way to implement a solver for formulas in logic Enf is the following (assuming the input is instance structure $\mathcal{A}$ and formula Φ has n phases):

1. Let B_1 be the set of all structure $\mathcal{B}_1$ such that $\mathcal{B}_1$ expands the instance structure $\mathcal{A}$ and satisfies the first phase;
2. Let B_i, $i \geq 2$, be the set of all structure $\mathcal{B}_i$ such that $\mathcal{B}_i$ expands at least one structure in B_{i-1} and satisfies the i-th phase;
3. If the specification has n phases, the problem is satisfiable iff B_n is non-empty.

In order to develop a solver based on the above idea, one needs to modify a SAT solver to produce and return all the solutions for a given SAT instance. Having a SAT solver with this feature, one can compute sets $B_1, \cdots, B_n$, as described above, by using a grounding-based solver. Unfortunately, a SAT instance in the form of CNF may have an exponential number of solutions, and so, implementing such a solver is not computationally efficient.

Let assume we have specification S with n phases, and $\mathcal{A}$ be the instance structure:

1. Let B_1 be a set of structures such that $\mathcal{B}_1 \in B_1$ iff $\mathcal{B}_1$ expands the instance structure $\mathcal{A}$ and satisfies the first phase;
2. Let B_i, $2 \leq i \leq n$, be the set of structures such that $\mathcal{B}_i \in B_i$ iff $\mathcal{B}_i$ expands at least one structure in B_{i-1} and satisfies the i-th phase;
3. We use S^i to denote the problem specification obtained by removing the first i phases of S and defining the predicates defined in any of the first i phases of S as instance predicates in S^i;
4. Let $\mathcal{B}_i$ and $\mathcal{B'}_i$ be two structures in B_i. The problem defined by problem specification S^i, and instance structure $\mathcal{B}_i$ has at least one solution iff the problem defined by problem specification S^i, and instance structure $\mathcal{B'}_i$ has at least one solution.

For a problem specification satisfying condition 4, given an instance structure, Enfragmo finds a solution iff the problem has a model based on the semantic of logic Enf.

The following specification rewrites the problem specification in Example 7 such that the new specification satisfies the above properties:

1. We modify the first phase of S, by adding another axiom to it: $\exists x\ E_1(x) \wedge \forall x\ E_1(x)$.

2. We do not need to change the second phase: $\forall x\ E_1(x) \wedge E_2(x)$.

Chapter 5

Grounding Techniques For Enfragmo

In Chapter 3, we described the input language of Enfragmo. As we explained, Enfragmo converts its inputs to a SAT instance in two phases. In this chapter and the next, we mainly focus on the first phase, the grounding phase, in which Enfragmo generates an equivalent variable-free first-order formula for a given problem.

5.1 Introduction

The results of the experiments in Chapter 3 showed that the performance of Enfragmo varies for different specifications of the same problem. Sometimes, the specifications have certain features that cause one grounding algorithm to fail, while allowing another algorithm to perform well. MXG [56], a solver developed by Mohebali as part of her Master thesis, and Enfragmo are both built based on the bottom-up grounding paradigm. However in Enfragmo, we have developed several new techniques/algorithms with the goal of accelerating the process of grounding and to shortening the solving phase.

In this chapter, we present the bottom-up relational algebra-based grounding algorithm, introduced in [58], as a *generic framework*, and explain the new techniques we have developed in Enfragmo, as a particular instantiation of this framework. Each of these configurations is suitable for certain kinds of specifications. We describe the properties of specifications for which each configuration is expected to perform well.

The grounding technique presented in this chapter only works on specifications where all the terms are variables and constants, i.e., the specifications cannot have aggregates, functions or arithmetical operators as their terms. In Chapter 6, we extend this grounding technique so that it supports more complex terms.

The rest of this chapter is organized as follows: In Section 5.2, the *generic framework* of relational algebra-based grounding is introduced. We describe the grounding algorithm based on an abstract notion of a table that supports certain operations. Section 5.3 describes several representations/data structures for tables, and discusses the advantages of each one in comparison with the others. We describe algorithms to implement the operations for each of the representations, in Section 5.4. In Section 5.5, we describe the experiments performed to confirm our claims regarding the relation between the properties of specifications and configurations.

5.1.1 My Contributions

The description of the relational algebra-based grounding algorithm, introduced in [58], as a generic framework is contributed by the author. Normal table, Subsection 5.3.2, and tables with hidden variables, Subsection 5.3.4, have been introduced in MXG [56]. Extending the idea of table with hidden variables to work with other tables is proposed by the author. True/False tables, Subsection 5.3.3, is based on [5]. Tables with restriction, Subsection 5.3.5 is designed and implemented by the author.

The sorting-based algorithms for joining and negating tables are proposed by Shahab Tasharrofi and the author. Also, the linear sorting algorithm for sorting the contents of tables is contributed by the author.

5.2 Relational Algebra-based Grounding

In [58], an algorithm that constructs a ground formula by a bottom-up process is introduced. This algorithm is based on an extension of the relational algebra [23]. In this section, we generalize this grounding algorithm to obtain a framework which allows us to handle grounding more efficiently.

To familiarize the reader with the concepts introduced in [58], and to distinguish our proposed generalization from the previous work, we briefly explain how the grounding algorithm introduced in [58] works. We describe, in more detail, how this algorithm handles the task of grounding in Subsection 7.1.1.

Given formula ϕ over vocabulary $\sigma \cup \varepsilon$ and σ-structure $\mathcal{A} = (A; \sigma^{\mathcal{A}})$, the goal of a grounding algorithm is to obtain an equivalent first-order variable-free formula ψ. In this thesis, we bring domain elements, A, 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 by $\tilde{A}$.

Definition 10 (Reduced Grounding for MX) *([58]) Formula ψ is a* reduced grounding *of formula ϕ over σ-structure $\mathcal{A} = (A, \sigma^{\mathcal{A}})$ if (1) ψ is a ground formula over $\varepsilon \cup \tilde{A}$; and (2) for every expansion structure $\mathcal{B} = (A, \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ over $vocab(\phi)$, $\mathcal{B} \models \phi$ iff $(\mathcal{B}, \tilde{A}) \models \psi$.*

An extended X-relation is a relation associated with the assignments to the tuple of variables in set X. The grounding algorithm produces/maintains extended relations. Notice that assignment $\gamma : X \mapsto A$ can be seen as a tuple of size $|X|$, whose elements are from A. In [58], an extended X-relation is defined as follows.

Definition 11 (extended X-relation; function $\delta_{\mathcal{R}}$) *([58]) 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 $\gamma : X \to A$; (2) for every γ, there is at most one ψ s.t. $(\gamma, \psi) \in \mathcal{R}$. The* function represented by $\mathcal{R}$*, is a mapping from tuples γ of elements of domain A to formulas, defined as:*

$$\delta_{\mathcal{R}}(\gamma) = \begin{cases} \psi & \text{if } (\gamma, \psi) \in \mathcal{R}, \\ \bot & \text{if there is no pair } (\gamma, \psi) \in \mathcal{R}. \end{cases}$$

Let γ be an assignment, then we use $\phi[\gamma]$ to denote the result of instantiating free variables in ϕ according to γ.

Definition 12 (answer to ϕ with respect to $\mathcal{A}$) *([58]) Let ϕ be a formula in $\sigma \cup \varepsilon$ with free variables X, $\mathcal{A}$ be a σ-structure with domain A, and $\mathcal{R}$ an extended X-relation over $\mathcal{A}$. If $\mathcal{R}$ is* an answer to ϕ with respect to $\mathcal{A}$*, for any $\gamma : X \to A$, we have $\delta_{\mathcal{R}}(\gamma)$ is a reduced grounding of $\phi[\gamma]$ over $\mathcal{A}$.*

The standard relational algebra has the following operations, each corresponding to a connective in FO: complement (negation); join (conjunction); union (disjunction), project (existential quantification); divide (universal quantification). In [58], the standard relation algebra is extended to work on extended relations.

Definition 13 (extended relational algebra operations) *([58]) Let $\mathcal{R}$ be an extended X-relation, and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *$\neg\mathcal{R}$ is the extended X-relation $\neg\mathcal{R} = \{(\gamma, \psi) \mid \gamma : X \to A, \delta_{\mathcal{R}}(\gamma) \neq \top,$ and $\psi = \neg\delta_{\mathcal{R}}(\gamma)\}$;*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is the extended $X \cup Y$-relation $\{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in \mathcal{R}, \gamma|_Y \in \mathcal{S},$ and $\psi = \delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y)\}$;*
3. *$\mathcal{R} \cup \mathcal{S}$ is the extended $X \cup Y$-relation $\mathcal{R} \cup \mathcal{S} = \{(\gamma, \psi) \mid \gamma|_X \in \mathcal{R}$ or $\gamma|_Y \in \mathcal{S},$ and $\psi = \delta_{\mathcal{R}}(\gamma|_X) \vee \delta_{\mathcal{S}}(\gamma|_Y)\}$;*
4. *The Y-*projection *of $\mathcal{R}$, denoted by $\pi_Y(\mathcal{R})$, is the extended Y-relation $\{(\gamma', \psi) \mid \gamma' = \gamma|_Y$ for some $\gamma \in \mathcal{R}$ and $\psi = \bigvee_{\{\gamma \in \mathcal{R} \mid \gamma' = \gamma|_Y\}} \delta_{\mathcal{R}}(\gamma)\}$;*
5. *The Y-*quotient *of $\mathcal{R}$, denoted by $d_Y(\mathcal{R})$, is the extended Y-relation $\{(\gamma', \psi) \mid$ for all γ such that $\gamma' = \gamma|_Y \Rightarrow \gamma \in \mathcal{R}$ and $\psi = \bigvee_{\{\gamma \in \mathcal{R} \mid \gamma' = \gamma|_Y\}} \delta_{\mathcal{R}}(\gamma)\}$;*
   $\{(\gamma', \psi) \mid \forall \gamma : X \to A \wedge \gamma|_Y = \gamma' \Rightarrow \gamma \in \mathcal{R}),$ and $\psi = \bigwedge_{\{\gamma \in \mathcal{R} \mid \gamma' = \gamma|_Y\}} \delta_{\mathcal{R}}(\gamma)\}$.

Proposition 5 *([58]) Suppose that $\mathcal{R}$ is an answer to ϕ_1 and $\mathcal{S}$ is an answer to ϕ_2, both with respect to structure $\mathcal{A}$.*

1. *$\neg\mathcal{R}$ is an answer to $\neg\phi_1$ with respect to $\mathcal{A}$;*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ with respect to $\mathcal{A}$;*
3. *$\mathcal{R} \cup \mathcal{S}$ is an answer to $\phi_1 \vee \phi_2$ with respect to $\mathcal{A}$;*
4. *If Y is the set of free variables of $\exists \bar{z} \phi_1$, then $\pi_Y(\mathcal{R})$ is an answer to $\exists \bar{z} \phi_1$ with respect to $\mathcal{A}$;*
5. *If Y is the set of free variables of $\forall \bar{z} \phi_1$, then $d_Y(\mathcal{R})$ is an answer to $\forall \bar{z} \phi_1$ with respect to $\mathcal{A}$.*

By applying the algebra inductively on the structure of the formula, we can obtain an answer to a given formula.

5.2.1 Generalization of Relational Algebra-based Grounding Technique

Patterson et al., in [58], define ψ as a reduced grounding for ϕ over σ-structure $\mathcal{A}$ if ψ is a grounding of ϕ over structure $\mathcal{A}$ and ψ does not have any symbol of σ. They also define the δ function of an extended X-relation, where X is a set of variables, to be a total function mapping instantiations of variables in X to reduced ground formulas. During the grounding process, the return value of function δ, for any extended relation, is uniquely determined by Definition 13.

In this section, we extend the above definitions to describe several alternatives to what has been proposed in [58]. We define an X-relation to be a mapping from instantiations of X to formulas, rather than a mapping from instantiations to reduced groundings.

Definition 14 (extended X-relation; function $\delta_{\mathcal{R}}$) *Let A be a domain, and X a set of variables. An* extended X-relation $\mathcal{R}$ *over A is a set of pairs (γ, ψ) such that,*
1) $\gamma : X \to A$;
2) ψ is a formula without any free variables;
3) for every γ, there is exactly one ψ such that $(\gamma, \psi) \in \mathcal{R}$.

The function $\delta_{\mathcal{R}}$ represented by $\mathcal{R}$ *is a mapping from object assignments to formulas, i.e.,*

$$\delta_{\mathcal{R}}(\gamma) = \psi \text{ if } (\gamma, \psi) \in \mathcal{R}.$$

Definition 15 (equivalent with respect to $\mathcal{A}$) *Let $\mathcal{A}$ be a σ-structure, $\phi_1(\bar{x})$ and $\phi_2(\bar{y})$ be two first-order formulas, where $\sigma \subseteq vocab(\phi_1)$ and $\sigma \subseteq vocab(\phi_2)$. We say ϕ_1 and ϕ_2 are equivalent with respect to $\mathcal{A}$ (or simply ϕ_1 and ϕ_2 are equivalent, if $\mathcal{A}$ is clear from the context) iff for any structure $\mathcal{B}$ expanding $\mathcal{A}$, we have*

$$\mathcal{B} \models \forall \bar{z}\ \phi_1 \leftrightarrow \phi_2,$$

where $\bar{z} = \bar{x} \cup \bar{y}$.

Example 8 *Let $\phi_1(x)$ be $E(x) \vee I(x)$, $\phi_2(x)$ be $E(x)$, and $\mathcal{A}$ be an $\{I\}$-structure. Also let $I^{\mathcal{A}} = \emptyset$. Formulas ϕ_1 and ϕ_2 are equivalent with respect to $\mathcal{A}$ since for any $\{I, E\}$-structure $\mathcal{B}$ expanding $\mathcal{A}$, we have $\mathcal{B} \models \forall x\ (I(x) \wedge E(x)) \leftrightarrow E(x)$.*

Example 9 *Let $\phi_1(x)$ and $\phi_2(x)$ be the same as Example 8, and let $\mathcal{A}$ be an $\{I\}$-structure such that $I^{\mathcal{A}} = \{1\}$. Let $\{I, E\}$-structure $\mathcal{B}$ be such that $I^{\mathcal{B}} = I^{\mathcal{A}} = \{1\}$ and $E^{\mathcal{B}} = \{1, 2\}$. Clearly, $\mathcal{B}$ expands $\mathcal{A}$ but $\mathcal{B} \not\models \forall x\,(I(x) \wedge E(x)) \leftrightarrow E(x)$. So, ϕ_1 and ϕ_2 are not equivalent with respect to $\mathcal{A}$.*

Note that if ϕ_1 and ϕ_2 are logically equivalent formulas, they are equivalent with respect to $\mathcal{A}$ for any structure $\mathcal{A}$.

Definition 16 (answer to ϕ with respect to $\mathcal{A}$) *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 $\mathcal{R}$ is* an answer to ϕ with respect to $\mathcal{A}$ *if for any $\gamma : X \to A$, we have $\delta_{\mathcal{R}}(\gamma)$ and a reduced grounding of $\phi[\gamma]$ are equivalent with respect to $\mathcal{A}$.*

Definition 16 is a generalization of Definition 12, used in [58]. In the latter, the return value of function δ must be a reduced grounding while in the former, we just need the output value of function δ to be equivalent to a reduced grounding of ϕ, with respect to the instance structure.

Since a sentence has no free variables, the answer to a sentence ϕ is a zero-ary extended X-relation, containing a single pair $(\langle\rangle, \psi)$, associating the empty tuple with formula ψ, which is equivalent to a (reduced) grounding of ϕ, with respect to the instance structure.

Following [58, 56], we explain the properties of extended relations corresponding to the outputs of these operators.

Definition 17 (Extended Relational Algebra) *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *$\neg\mathcal{R}$ is an extended X-relation such that for $\gamma : X \mapsto A$, $\delta_{\neg\mathcal{R}}(\gamma)$ and $\neg\delta_{\mathcal{R}}(\gamma)$ are equivalent with respect to $\mathcal{A}$.*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is an extended $X \cup Y$-relation such that for $\gamma : X \mapsto A$, $\delta_{\mathcal{R}\bowtie\mathcal{S}}(\gamma)$ and $\delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y)$ are equivalent with respect to $\mathcal{A}$.*
3. *$\mathcal{R} \cup \mathcal{S}$ is an extended $X \cup Y$-relation such that for $\gamma : X \mapsto A$, $\delta_{\mathcal{R}\cup\mathcal{S}}(\gamma)$ and $\delta_{\mathcal{R}}(\gamma|_X) \vee \delta_{\mathcal{S}}(\gamma|_Y)\}$ are equivalent with respect to $\mathcal{A}$.*
4. *For $Z \subseteq X$, the Z-*projection *of $\mathcal{R}$, denoted by $\pi_Z(\mathcal{R})$, is an extended Z-relation such that for all $\gamma : Z \mapsto A$, $\delta_{\pi_Z(\mathcal{R})}(\gamma)$ and $\bigvee_{\{\gamma' : X\mapsto A|\gamma=\gamma'|_Z\}} \delta_{\mathcal{R}}(\gamma')$ are equivalent with respect to $\mathcal{A}$.*
5. *For $Z \subseteq X$, the Z-*quotient *of $\mathcal{R}$, denoted by $d_Z(\mathcal{R})$, is an extended Z-relation such that for all $\gamma : Z \mapsto A$, $\delta_{d_Z(\mathcal{R})}(\gamma)$ and $\bigwedge_{\{\gamma' : X\to A|\gamma=\gamma'|_Z\}} \delta_{\mathcal{R}}(\gamma')$ are equivalent with respect to $\mathcal{A}$.*

Our definition of extended relational algebra, Definition 17, is a generalization of what has been proposed previously. Unlike Definition 13, our definition simply describes the necessary and sufficient properties the output value of function δ must have, for our extended relational algebra to work correctly.

In [58], the answer to atomic formula $P(\bar{x})$ is defined as follows: If P is an instance predicate, the answer to P is defined as the extended relation corresponding to set $\{(\bar{a}, \top) \mid \text{ if } \bar{a} \in P^{\mathcal{A}}\} \cup \{(\bar{a}, \bot) \mid \text{ if } \bar{a} \notin P^{\mathcal{A}}\}$. If P is an expansion predicate, the answer is the set of all pairs $(\bar{a}, P(\bar{a}))$, where $\bar{a}$ is a tuple of elements from domain A. The atomic formulas of the form $x_1\{\leq, <, =, >, \geq\}x_2$, where x_1 and x_2 are two variables, can be seen as instance predicates since one can evaluate their truth values for any given assignment.

Definition 18 (Answer to Atomic Formula) *Let ϕ be an atomic formula with free variables $X = \{x_1, \cdots, x_n\}$ and $\mathcal{R}$ be an extended X-relation.*

1. *Let ϕ be $P(x_1, \cdots, x_n)$ where P is an n-ary instance predicate. Extended relation $\mathcal{R}$ is an answer to ϕ iff for $\gamma : X \mapsto A$, $\delta_{\mathcal{R}}(\gamma)$ and $P^{\mathcal{A}}[\gamma]$ are equivalent with respect to $\mathcal{A}$.*

2. *Let ϕ be $P(x_1, \cdots, x_n)$ where P is an n-ary expansion predicate. Extended relation $\mathcal{R}$ is an answer to ϕ iff for $\gamma : X \mapsto A$, $\delta_{\mathcal{R}}(\gamma)$ and $P[\gamma]$ are equivalent with respect to $\mathcal{A}$.*

3. *Let ϕ be x_1 op x_2 where op is a comparison operator. Extended relation $\mathcal{R}$ is an answer to ϕ iff for $\gamma : X \mapsto A$, $\delta_{\mathcal{R}}(\gamma)$ and $x_1|_\gamma$ op $x_2|_\gamma$ are equivalent with respect to $\mathcal{A}$.*

In the context of this thesis, the above definition describes just one of the approaches to defining the answer to an atomic formula. We propose different approaches for representing answers to the atomic formulas in the next sections.

To ground using this algebra, we apply the algebra inductively on the structure of the formula, just as the standard relational algebra may be applied for query evaluation. The correctness of the method then follows, by induction on the structure of the formula, from the following proposition.

Proposition 6 *Suppose that $\mathcal{R}$ is an answer to ϕ_1 and $\mathcal{S}$ is an answer to ϕ_2, both over domain A. Then*

1. *$\neg\mathcal{R}$ is an answer to $\neg\phi_1$ with respect to $\mathcal{A}$;*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ with respect to $\mathcal{A}$;*
3. *$\mathcal{R} \cup \mathcal{S}$ is an answer to $\phi_1 \vee \phi_2$ with respect to $\mathcal{A}$;*

4. *If Y is the set of free variables of $\exists \bar{z}\phi_1$, then $\pi_Y(\mathcal{R})$ is an answer to $\exists \bar{z}\phi_1$ with respect to $\mathcal{A}$;*
5. *If Y is the set of free variables of $\forall \bar{z}\phi_1$, then $d_Y(\mathcal{R})$ is an answer to $\forall \bar{z}\phi_1$ with respect to $\mathcal{A}$.*

Proof: Here, we prove $\mathcal{T}$, where $\mathcal{T} = \mathcal{R} \bowtie \mathcal{S}$, is an answer to $\phi_1 \wedge \phi_2$ with respect to $\mathcal{A}$. The proofs of the other claims are similar to this one.

We have $\mathcal{R}$ is an answer to $\phi_1(X)$ with respect to $\mathcal{A}$, and $\mathcal{S}$ is an answer to $\phi_2(Y)$ with respect to $\mathcal{A}$. Based on Definition 16, for all $\gamma : X \cup Y \mapsto A$, and structure $\mathcal{B}$ expanding $\mathcal{A}$, we have $(\mathcal{B}, \tilde{A}) \models \delta_{\mathcal{R}}(\gamma|_X)$ iff $(\mathcal{B}, \tilde{A}) \models \phi_1[\gamma|_X]$. Also, we have $(\mathcal{B}, \tilde{A}) \models \delta_{\mathcal{S}}(\gamma|_Y)$ iff $(\mathcal{B}, \tilde{A}) \models \phi_2[\gamma|_Y]$.

Let ψ be a reduced grounding for $(\phi_1(X) \wedge \phi_2(Y))\,[\gamma]$ over $\mathcal{A}$. Then, for all $\mathcal{B} = (A, \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$, $(\mathcal{B}, \tilde{A}) \models \psi$ iff $(\mathcal{B}, \tilde{A}) \models (\phi_1(X) \wedge \phi_2(Y))\,[\gamma]$. From $(\mathcal{B}, \tilde{A}) \models (\phi_1(X) \wedge \phi_2(Y))\,[\gamma]$, we conclude $(\mathcal{B}, \tilde{A}) \models (\phi_1[\gamma|_X] \wedge \phi_2[\gamma|_Y])$. Since $\mathcal{R}$ ($\mathcal{T}$) is an answer to ϕ_1 (ϕ_2, respectively) with respect to $\mathcal{A}$, we have $(\mathcal{B}, \tilde{A}) \models (\phi_1(X) \wedge \phi_2(Y))[\gamma]$ iff $(\mathcal{B}, \tilde{A}) \models (\delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y))$. According Definition 17, $\delta_{\mathcal{T}}(\gamma)$ and $(\delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{R}}(\gamma|_Y))$ are equivalent, so $(\mathcal{B}, \tilde{A}) \models (\phi_1(X) \wedge \phi_2(Y))[\gamma]$ iff $(\mathcal{B}, \tilde{A}) \models \delta_{\mathcal{T}}(\gamma)$.

To make the proofs easier to read, in the other proofs of this chapter, we may use $\mathcal{B}$ instead of $(\mathcal{B}, \tilde{A})$.

■

Algorithm 1 represents the grounding algorithm using the extended relational algebra technique.

Algorithm 1 Relational Algebra-based Grounding Algorithm([56]).

Input: ϕ, A, $\sigma^{\mathcal{A}}$
Output: $\mathcal{R}$: $\mathcal{R}$ is an answer to Φ.
1: **function** GROUND(ϕ, A, $\sigma^{\mathcal{A}}$.)
2: **if** $\phi = \neg\phi_1$ **then**
3: **return** $\neg$ GROUND($\phi_1, A, \sigma^{\mathcal{A}}$)
4: **else if** $\phi = \phi_1 \wedge \phi_2$ **then**
5: **return** GROUND($\phi_1, A, \sigma^{\mathcal{A}}$) $\bowtie$ GROUND($\phi_2, A, \sigma^{\mathcal{A}}$)
6: **else if** $\phi = \phi_1 \vee \phi_2$ **then**
7: **return** GROUND($\phi_1, A, \sigma^{\mathcal{A}}$) $\cup$ GROUND($\phi_2, A, \sigma^{\mathcal{A}}$))
8: **else if** $\phi = \exists\bar{z}\phi_1$ **then**
9: **return** $\pi_{\bar{z}}$ (GROUND($\phi_1, A, \sigma^{\mathcal{A}}$))
10: **else if** $\phi = \forall\bar{z}\phi_1$ **then**
11: **return** $d_{\bar{z}}$ (GROUND($\phi_1, A, \sigma^{\mathcal{A}}$))
12: **else if** ϕ is an instance predicate **then**
13: **return** GROUNDINSTANCEPREDICATE($\phi, A, \sigma^{\mathcal{A}}$))
14: **else**
15: **return** GROUNDEXPANSIONPREDICATE($\phi, A, \sigma^{\mathcal{A}}$)
Input: $\phi = P(\bar{x})$ where P is an instance predicate, A, $\sigma^{\mathcal{A}}$
Output: $\mathcal{R}$: $\mathcal{R}$ is an answer to $P(\bar{x})$
16: **function** GROUNDINSTANCEPREDICATE(ϕ, A, $\sigma^{\mathcal{A}}$)
Construct $\mathcal{R}$ such that, for every instantiation $\gamma : X \mapsto A$,
$\delta_{\mathcal{R}}(\gamma)$ is equivalent to $P^{\mathcal{A}}[\gamma]$.
Input: $\phi = P(\bar{x})$ where P is an expansion predicate, A, $\sigma^{\mathcal{A}}$
Output: $\mathcal{R}$: $\mathcal{R}$ is an answer to $P(\bar{x})$
17: **function** GROUNDEXPANSIONPREDICATE(ϕ, A, $\sigma^{\mathcal{A}}$)
Construct $\mathcal{R}$ such that, for every instantiation $\gamma : X \mapsto A$,
$\delta_{\mathcal{R}}(\gamma)$ is equivalent to $\phi[\gamma]$.
Input: $\phi = X_1\ op\ X_2$ where op is a comparison operator, A, $\sigma^{\mathcal{A}}$
Output: $\mathcal{R}$: $\mathcal{R}$ is an answer to ϕ
18: **function** GROUNDORDERINGFORMULA(ϕ, A, $\sigma^{\mathcal{A}}$)
Construct $\mathcal{R}$ such that, for every instantiation $\gamma : X \mapsto A$,
$\delta_{\mathcal{R}}(\gamma)$ is equivalent to $\phi[\gamma]$.

5.3 Different Representations for Tables

The number of tuples in an extended X-relation is $|A|^k$, where $k = |X|$. In this section, we describe different approaches to provide a compact representation of an extended relation, which may have a huge size. In the next section, we explain how each of the relational algebra operators efficiently computes the resulting extended relations given each of these representations.

The rest of this section is divided into subsections, in each of which we describe a different approach for representing an extended X-relation. In each subsection, we first give a high level

description of the idea and discuss its cons and pros, and then we describe how the result of relational algebra operators can be computed. In Section 5.4, we discuss the most suitable data structure to implement each representation and explain the algorithms to compute the result of operators on each data structure.

Three of these representations (basic tables, normal tables and tables with hidden variables) are proposed and implemented in [56] and the rest are new. In the setting of [56], to describe a representation, one needs to introduce a new algebra. On the other hand, the generalization of the relational algebra, introduced in the previous section, enables us to describe a representation as a data structure. To define a representation, we just need to describe how the output of function δ can be computed and how the extended relation resulting from the application of a relational algebra operator can be represented.

For the sake of explanation, we restrict the Boolean operators in the axiomatizations to $\neg, \wedge$ and the quantifiers to existential quantifier ($\exists$), and hence, we need to describe how operators $\neg, \bowtie, \Pi$ can be implemented for different kinds of representations.

In this chapter, we use R^s_X to denote the table representing extended X-relation, where the superscript, s, can be either "B", "N", "TF", "H", "r" (The superscript identifies the table kind). To avoid introducing new notation, we use δ, $\neg$, $\bowtie$, Π to denote the delta function, not, join and projection operators, respectively, on the tables.

5.3.1 Basic Table

The naive approach to represent an extended X-relation is to use a set of pairs (γ, ψ), where γ is a tuple of elements and ψ is a formula. One can think of this set as a table with $|X| + 1$ columns and $|A|^{|X|}$ rows in which the i-th column, $i \leq |X|$, corresponds to the i-th variable in X and the last column is the formula column. This is the motivation behind calling the data structures used to represent extended X-relations, tables.

Let R^B_X denote a basic table representing an extended X-relation $\mathcal{R}$. We use $T(R^B_X)$ to denote the set of tuples stored in this data structure. Clearly, set $T(R^B_X)$ contains $|A|^{|X|}$ elements and function $\delta_{R^B_X}$ is computed as:

$$\delta_{R^B_X}(\gamma) = \psi[\gamma] \text{ if } (\gamma, \psi) \in T(R^B_X).$$

Construction 1 describes one method of representing the result of applying relational algebra operators defined in Definition 17 on basic tables.

Construction 1 *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *Let $\mathcal{T}$ be an extended X-relation defined as $\neg\mathcal{R}$. The following is used to construct a basic table for $\mathcal{T}$.*
   $T(T^B_X) = \{(\gamma, \neg\psi) \mid (\gamma, \psi) \in T(R^B_X)\}$;
2. *Let $\mathcal{T}$ be an extended $X \cup Y$-relation defined as $\mathcal{R} \bowtie \mathcal{S}$. The following is used to construct a basic table for $\mathcal{T}$.*
   $T(T^B_{X\cup Y}) = \{(\gamma, \psi_1 \wedge \psi_2) \mid (\gamma|_X, \psi_1) \in T(\mathcal{R}^B_X)$ *and* $(\gamma|_Y, \psi_2) \in T(\mathcal{S}^B_Y)\}$.

3. *Let $\mathcal{T}$ be an extended Z-relation defined as $\pi_Z(\mathcal{R})$. The following is used to construct a basic table for $\mathcal{T}$.*
   $T(T_Z^B) = \{(\gamma, \psi) \mid \psi = \bigvee_{\{\gamma' : X \to A \mid \gamma = \gamma'|_Z\}} \delta_{R_X^B}(\gamma')[\gamma'|_{X \setminus Z}]\}$.

Proposition 7 *Construction 1 is correct, that is, for each operator, the construction generates a basic table whose δ function satisfies the properties stated in Definition 16.*

Proof: Here, we provide the proof for the negation operator. The proofs for the other operators are similar.

We show for all γ, $\delta_{T_X^B}(\gamma) = \neg\delta_{R_X^B}(\gamma)$.

Let $T(R_X^B)$ be the set of tuples in basic table R_X^B and $\gamma : X \mapsto A$ and $\phi = \delta_{R_X^B}(\gamma)$. Since $\delta_{R_X^B}(\gamma) = \phi$, we conclude the pair (γ, ϕ) is in set $T(R_X^B)$. From Construction 1, we have $(\gamma, \neg\phi) \in T(T_X^B)$, and then $\delta_{T_X^B}(\gamma)$ evaluates as $\neg\phi$. So, we have $\delta_{T_X^B}(\gamma) = \neg\phi = \neg\delta_{R_X^B}(\gamma)$.

■

A basic table for atomic formula $\phi(\bar{x})$ has $|A|^{|\bar{x}|}$ rows where the formula attached to the row with assignment γ is a formula equivalent to $\phi[\gamma]$.

To illustrate the difference between the representations, we use Example 10 as a running example.

Example 10 *(Running example) Let $\sigma = \{P\}$ and $\varepsilon = \{Q\}$ where both P and Q are binary predicates. Also let $\mathcal{A} = (A; P^{\mathcal{A}})$, $A = \{1, 2, 3\}$ and $P^{\mathcal{A}} = \{(2, 1), (2, 2), (2, 3), (3, 1)\}$. We study how different data structures represent an answer to each of the following formulas:*

- $\phi_1(x, y) = P(x, y)$,
- $\phi_2(x, y) = Q(x, y)$,
- $\phi_3(x, y) = \neg\phi_1(x, y)$,
- $\phi_4(x, y, z) = \phi_2(x, y) \wedge \phi_3(y, z)$,
- $\phi_5(y) = \exists x\ z\ \phi_4(x, y, z)$.

Here, we illustrate how an answer to each of the above formulas, given the instance structure $\mathcal{A}$, can be represented as a Basic table.

x	y	$\psi(x, y)$
1	1	$\bot$
1	2	$\bot$
1	3	$\bot$
2	1	$\top$
2	2	$\top$
2	3	$\top$
3	1	$\top$
3	2	$\bot$
3	3	$\bot$

Table 5.1: Basic table for $\phi_1(x, y)$.

x	y	$\psi(x, y)$
1	1	$Q(x, y)$
1	2	$Q(x, y)$
1	3	$Q(x, y)$
2	1	$Q(x, y)$
2	2	$Q(x, y)$
2	3	$Q(x, y)$
3	1	$Q(x, y)$
3	2	$Q(x, y)$
3	3	$Q(x, y)$

Table 5.2: Basic table for $\phi_2(x, y)$.

x	y	$\psi(x, y)$
1	1	$\top$
1	2	$\top$
1	3	$\top$
2	1	$\bot$
2	2	$\bot$
2	3	$\bot$
3	1	$\bot$
3	2	$\top$
3	3	$\top$

Table 5.3: Basic table for $\phi_3(x, y)$.

x	y	z	$\psi(x,y,z)$
1	1	1	$Q(x,y)$
1	1	2	$Q(x,y)$
1	1	3	$Q(x,y)$
1	2	1	$\bot$
⋮	⋮	⋮	

Table 5.4: Basic table for $\phi_4(x,y,z)$.

y	$\psi(y)$
1	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
2	$\bot$
3	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$

Table 5.5: Basic table for $\phi_5(y)$.

5.3.2 Normal Table

The first observation we use to improve the performance of basic tables is that almost all instance predicates have sparse interpretations, i.e., there are many instantiations whose corresponding formulas are *false* ($\bot$). For a given extended X-relation, if we remove all tuples whose formula parts are $\bot$ and store the rest in a table (or an array), we save a large amount of space. Then, to obtain the corresponding formula for an instantiation, we search for that instantiation in the table. If such an instantiation has been found, we return the corresponding formula, otherwise that instantiation is mapped to $\bot$ [56].

More formally, let R_X^N be a normal table representing an extended X-relation $\mathcal{R}$. We use $T(R_X^N)$ to denote the set of tuples stored in this data structure. Function $\delta_{R_X^N}$, for this representation, is defined as:

$$\delta_{R_X^N}(\gamma) = \begin{cases} \psi[\gamma] & \text{if } (\gamma,\psi) \in T(R_X^N), \\ \bot & \text{if there is no pair } (\gamma,\psi) \in T(R_X^N). \end{cases}$$

In [56], $\delta_{R_X^N}(\gamma)$ is defined as:

$$\delta_{R_X^N}(\gamma) = \begin{cases} \psi & \text{if } (\gamma,\psi) \in T(R_X^N), \\ \bot & \text{if there is no pair } (\gamma,\psi) \in T(R_X^N). \end{cases}$$

Note that the difference is when $(\gamma,\psi) \in T(\binom{N}{X})$; our normal table returns $\psi[\gamma]$ while the normal table defined in [56] returns ψ. The way $\delta_{R_X^N}$ is defined in the latter requires creating a formula for each instantiation γ. This method is not efficient and consumes a large amount of memory. As can be seen in Example 11, our definition of $\delta_{R_X^N}$ for normal tables, allows us to use the same ψ in different rows, while [56] does not.

Construction 2 describes how the normal tables resulting from applying relational algebra operators, defined in Definition 17, are computed.

Construction 2 *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *Let $\mathcal{T}$ be an extended X-relation defined as $\neg\mathcal{R}$. The following is used to construct a normal table representing $\mathcal{T}$.*
   $T(T_X^N) = \{(\gamma,\neg\psi) \mid (\gamma,\psi) \in T(R_X^N), \psi \neq \top\} \cup \{(\gamma,\top) \mid \gamma : \delta_{R_X^N}(\gamma) = \bot\}$;

2. *Let $\mathcal{T}$ be an extended $X \cup Y$-relation defined as $R \bowtie S$. The following is used to construct a normal table representing $\mathcal{T}$.*
   $T(T_X^N) = \{(\gamma, \psi_1 \wedge \psi_2) \mid (\gamma|_X, \psi_1) \in T(R_X^N)$ *and* $(\gamma|_Y, \psi_2) \in T(S_Y^N)\}$;
3. *Let T_Z^N be the table computed as $\pi_Z(R_X^N)$. The following is used to construct a normal table representing T_Z^N.*
   $T(T_Z^N) = \{(\gamma, \psi) \mid$ *there is at least one* γ' *s.t.* $(\gamma', \psi') \in T(R_X^N)$ *and* $\gamma = \gamma'|_Z$, $\psi = \bigvee_{\{\gamma' : X \mapsto A \text{ such that } \gamma = \gamma'|_Z\}} \delta_{R_X^N}(\gamma')[\gamma'|_{(X \setminus Z)}]\}$.

Proposition 8 *Construction 2 is correct, that is, for each operator, the construction generates a normal table whose δ function satisfies the properties stated in Definition 16.*

Proof: Here, we only provide the proof for the negation operator. The proofs for the other operators are similar.

We consider the following three cases:

1. If $\delta_{R_X^N}(\gamma) = \bot$, $T(R_X^N)$ does not have any entry whose first component is γ, and hence, $\delta_{T_X^N}(\gamma) = \top$.

2. If $\delta_{R_X^N}(\gamma) = \top$, $T(R_X^N)$ contains the pair $\langle \gamma, \top \rangle$. By our construction, there will be no pair, in $T(T_X^N)$ whose first element is γ and so $\delta_{T_X^N}(\gamma) = \bot$.

3. If $\delta_{R_X^N}(\gamma) = \phi \notin \{\bot, \top\}$, $T(R_X^N)$ must have pair $\langle \gamma, \phi \rangle$ in it. Based on the construction, $T(T_X^N)$ will have $\langle \gamma, \neg\phi \rangle$, and so $\delta_{T_X^N}(\gamma) = \neg\phi$.

We show that $\delta_{T_X^N}(\gamma) \Leftrightarrow \neg\delta_{R_X^N}(\gamma)$, for all γ, which is, according to Definition 17, enough to complete the proof. ■

Example 11 *(Continuation of Example 10) Tables 5.6,...,5.10 represent answers in the form of normal tables for $\phi_1, \ldots, \phi_5$, in the setting of Example 10.*

x	y	$\psi(x,y)$
2	1	$\top$
2	2	$\top$
2	3	$\top$
3	1	$\top$

Table 5.6: Normal table for $\phi_1(x, y)$.

x	y	$\psi(x,y)$
1	1	$Q(x,y)$
1	2	$Q(x,y)$
1	3	$Q(x,y)$
2	1	$Q(x,y)$
2	2	$Q(x,y)$
2	3	$Q(x,y)$
3	1	$Q(x,y)$
3	2	$Q(x,y)$
3	3	$Q(x,y)$

Table 5.7: Normal table for $\phi_2(x, y)$.

x	y	$\psi(x,y)$
1	1	$\top$
1	2	$\top$
1	3	$\top$
3	2	$\top$
3	3	$\top$

Table 5.8: Normal table for $\phi_3(x, y)$.

x	y	z	$\psi(x,y,z)$
1	1	1	$Q(x,y)$
1	1	2	$Q(x,y)$
⋮	⋮	⋮	

Table 5.9: Normal table for $\phi_4(x,y,z)$.

y	$\psi(y)$
1	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
3	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$

Table 5.10: Normal table for $\phi_5(y)$.

Comparing Table 5.1 and Table 5.6 illustrates how a normal table can save memory when we are dealing with instance predicates.

5.3.3 True/False Tables

Although normal tables can represent some large extended relations compactly, there are extended relations whose corresponding normal tables have too many rows. In this subsection and the next, we describe two modifications to the normal tables that enable us to represent certain extended relations more compactly.

Normal tables compactly represent extended relations that have many rows with *false* formula attached to them. From another perspective, this type of table is a representation with *false* as its default formula. If an instantiation is not in a table, it is mapped to the default formula of the table. We can extend this idea and define two types of tables; the False Table, whose default formula is *false*, and the True Table, whose default formula is *true*.

True/False table R_X^{TF} representing an extended X-relation $\mathcal{R}$ is described using a set of tuples (set of rows), $T(R_X^{TF})$, and a formula (default formula), $D(R_X^{TF})$. Function $\delta_{R_X^{TF}}$, for this representation, is defined as:

$$\delta_{R_X^{TF}}(\gamma) = \begin{cases} \psi[\gamma] & \text{if } (\gamma,\psi) \in T(R_X^{TF}), \\ D(R_X^{TF}) & \text{if there is no pair } (\gamma,\psi) \in T(R_X^{TF}). \end{cases}$$

Construction 3 *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *Let $\mathcal{T}$ be an extended X-relation defined as $\neg\mathcal{R}$. The following is used to construct a True/-False table for $\mathcal{T}$.*
   - *(a)* $T(T_X^{TF}) = \{(\gamma,\neg\psi) \mid (\gamma,\psi) \in T(R_X^{TF})\}$;
   - *(b)* $D(T_X^{TF}) = \neg D(R_X^{TF})$.
2. *Let $\mathcal{T}$ be an extended $X \cup Y$-relation defined as $\mathcal{R} \bowtie \mathcal{S}$. The following is used to construct a True/False table for $\mathcal{T}$.*

   If both R_X^{TF} and S_Y^{TF} are False tables:
   - *(a)* $T(T_{X\cup Y}^{TF}) = \{(\gamma,\psi) \mid (\gamma|_X,\phi_1) \in T(R_X^{TF})$ *and* $(\gamma|_Y,\phi_2) \in T(S_Y^{TF}), \psi = \phi_1 \wedge \phi_2\}$;
   - *(b)* $D(T_{X\cup Y}^{TF}) = \bot$.

   If both R_X^{TF} and S_Y^{TF} are True tables:

(a) $T(T^{TF}_{X\cup Y}) = \{(\gamma,\psi) \mid (\gamma|_X, \phi_1) \in T(R^{TF}_X)$ *and* $(\gamma|_Y, \phi_2) \in T(S^{TF}_Y), \psi = \phi_1 \wedge \phi_2\} \cup \{(\gamma,\psi) \mid (\gamma|_X, \psi) \in T(R^{TF}_X)$ *and* $\gamma|_Y \notin T(S^{TF}_Y)\} \cup \{(\gamma,\psi) \mid \gamma|_X \notin T(R^{TF}_X)$ *and* $(\gamma|_Y, \psi) \in T(S^{TF}_Y)\}$;

(b) $D(T^{TF}_{X\cup Y}) = \top$.

If R^{TF}_X is a False table and S^{TF}_Y a True table:

(a) $T(T^{TF}_{X\cup Y}) = \{(\gamma,\psi) \mid \gamma|_X \in T(R^{TF}_X)$ *and* $\gamma|_Y \in T(S^{TF}_Y)$ *and* $\delta_{S^{TF}_Y}(\gamma|_Y) \neq \bot, \psi = \delta_{R^{TF}_X}(\gamma|_X) \wedge \delta_{S^{TF}_Y}(\gamma|_Y)\} \cup \{(\gamma,\psi) \mid (\gamma|_X, \psi) \in T(R^{TF}_X)$ *and* $\gamma|_Y \notin T(S^{TF}_Y)\}$;

(b) $D(\mathcal{T}^{TF}) = false$.

3. *Let $\mathcal{T}^{TF}$ be an extended Z-relation defined as $\pi_Z(\mathcal{R}^{TF})$. The following is used to construct a True/False table for $\mathcal{T}$.*

R^{TF}_X is a False table:

(a) $T(T^{TF}_Z) = \{(\gamma,\psi) \mid \gamma :$ *there is at least one γ' s.t. $\gamma' \in T(R^{TF}_X)$ and $\gamma = \gamma'|_Z$, $\psi = \bigvee_{\{\gamma': X\to A \mid \gamma = \gamma'|_Z\}} \delta_{R^{TF}_X}(\gamma')[\gamma'|_{(X\setminus Z)}]\}$;*

(b) $D(T^{TF}_Z) = D(R^{TF}_X)$.

R^{TF}_X is a True table:

(a) $T(T^{TF}_Z) = \{(\gamma,\psi) \mid \gamma :$ *for all $\gamma' : X \mapsto A$ s.t. $\gamma = \gamma'|_Z$, we have $\langle\gamma', \phi\rangle \in T(T^{TF}_X)$, $\psi = \bigvee_{\{\gamma': X\to A \mid \gamma = \gamma'|_Z\}}, \delta_{R^{TF}_X}(\gamma')[\gamma'|_{(X\setminus Z)}]\}$;*

(b) $D(T^{TF}_Z) = D(R^{TF}_X)$.

Proposition 9 *Construction 3 is correct, that is, for each operator, the construction generates a True/False table whose δ function satisfies the properties stated in Definition 16.*

Proof: Here, we provide the proof for the negation and join operators.

- Correctness of construction for negation operator: By Definition 17, it is enough to show $\delta_{T^{TF}_X}(\gamma) = \neg\delta_{R^{TF}_X}(\gamma)$, for all γ.

  We consider the following two cases:

  1. If $\delta_{R^{TF}_X}(\gamma) = D(R^{TF}_X)$, then $T(R^{TF}_X)$ does not have any entry whose first component is γ. Based on the construction, $T(T^{TF}_X)$ also does not have any entry whose first component is γ. So, we have $\delta_{T^{TF}_X}(\gamma) = D(T^{TF}_X) = \neg D(R^{TF}_X) = \neg\delta_{R^{TF}_X}(\gamma)$.
  2. If $\delta_{R^{RF}_X}(\gamma) = \phi \neq D(R^{TF}_X)$, then $\langle\gamma,\phi\rangle \in T(R^{TF}_X)$. Based on the construction, $\langle\gamma,\neg\phi\rangle \in T(T^{TF}_X)$. So $\delta_{T^{TF}_X}(\gamma) = \neg\phi = \neg\delta_{R^{TF}_X}(\gamma)$.

- Correctness of construction for join operator: We only present the proof for the case when one of the tables is a True table and the other one is a False table.

  By Definition 17, it is enough to show $\delta_{T^{TF}_{X\cup Y}}(\gamma) \Leftrightarrow \delta_{R^{TF}_X}(\gamma) \wedge \delta_{S^{TF}_Y}(\gamma)$, for all γ. We consider the following four cases:

1. If $\delta_{R_X^{TF}}(\gamma) = D(R_X^{TF})$ and $\delta_{S_Y^{TF}}(\gamma) = D(S_Y^{TF})$, then neither $T(R_X^{TF})$ nor $T(S_X^{TF})$ has any entry whose first component is γ. Based on the construction, $T(T_X^{TF})$ does not have any entry with the first component γ, either. So, we have $\delta_{T_{X\cup Y}^{TF}}(\gamma) = D(T_X^{TF}) = \bot \Leftrightarrow \bot \wedge \top = \delta_{R_X^{TF}}(\gamma) \wedge \delta_{S_Y^{TF}}(\gamma)$.
2. If $\delta_{R_X^{TF}}(\gamma) = \phi \neq D(R_X^{TF})$ and $\delta_{S_Y^{TF}}(\gamma) = D(S_Y^{TF})$, then we have $(\gamma, \phi) \in T(R_X^{TF})$. From the definition of δ function for True/False table, $\delta_{T_{X\cup Y}^{TF}}(\gamma) = \phi \Leftrightarrow \phi \wedge \top = \delta_{R_X^{TF}}(\gamma) \wedge \delta_{S_Y^{TF}}(\gamma)$.
3. If $\delta_{R_X^{TF}}(\gamma) = D(R_X^{TF})$ and $\delta_{S_Y^{TF}}(\gamma) = \psi \neq D(S_Y^{TF})$, then $T(R_X^{TF})$ does not have any entry whose first component is γ. Based on the construction, $T(T_{X\cup Y}^{TF})$ does not have any entry with the first component γ, either. So, we have $\delta_{T_{X\cup Y}^{TF}}(\gamma) = D(T_{X\cup Y}^{TF}) = \bot \Leftrightarrow \bot \wedge \psi = \delta_{R_X^{TF}}(\gamma) \wedge \delta_{S_Y^{TF}}(\gamma)$.
4. If $\delta_{R_X^{TF}}(\gamma) = \phi \neq D(R_X^{TF})$ and $\delta_{S_Y^{TF}}(\gamma) = \psi \neq D(S_Y^{TF})$, then by the construction, we have $(\gamma, \phi \wedge \psi) \in T(R_{X\cup Y}^{TF})$, which means $\delta_{T_{X\cup Y}^{TF}}(\gamma) = \phi \wedge \psi = \delta_{R_X^{TF}}(\gamma) \wedge \delta_{S_Y^{TF}}(\gamma)$.

In all four cases, formulas $\delta_{T_{X\cup Y}^{TF}}(\gamma)$ and $\delta_{R_X^{TF}}(\gamma) \wedge \delta_{S_Y^{TF}}(\gamma)$ are equivalent, which proves the proposition.

■

Example 12 *(Continuation of Example 10) Tables 5.11,...,5.15 represent answers in the form of True/False tables for $\phi_1, \ldots, \phi_5$, in the setting of Example 10.*

x	y	$\psi(x,y)$
2	1	$\top$
2	2	$\top$
2	3	$\top$
3	1	$\top$
$D = \bot$		

Table 5.11: True/False table for $\phi_1(x, y)$.

x	y	$\psi(x,y)$
1	1	$Q(x,y)$
1	2	$Q(x,y)$
1	3	$Q(x,y)$
2	1	$Q(x,y)$
2	2	$Q(x,y)$
2	3	$Q(x,y)$
3	1	$Q(x,y)$
3	2	$Q(x,y)$
3	3	$Q(x,y)$
$D = \bot$		

Table 5.12: True/False table for $\phi_2(x, y)$.

x	y	$\psi(x,y)$
2	1	$\bot$
2	2	$\bot$
2	3	$\bot$
3	1	$\bot$
$D = \top$		

Table 5.13: True/False table for $\phi_3(x, y)$.

x	y	z	$\psi(x,y,z)$
1	1	1	$Q(x,y)$
1	1	2	$Q(x,y)$
⋮	⋮	⋮	
$D = \bot$			

Table 5.14: True/False table for $\phi_4(x,y,z)$.

y	$\psi(y)$
1	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
3	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
$D = \bot$	

Table 5.15: True/False table for $\phi_5(y)$.

5.3.4 Table with Hidden variables

The table corresponding to the answer of atomic formula $P(\bar{x})$, where P is from the expansion vocabulary, is a table all of whose columns are universal; that is, the table contains all the possible $|A|^{|\bar{x}|}$ rows. In practice, we may represent this table implicitly and avoid enumerating all the tuples. As operations are applied, some columns remain universal, while others do not. The universal columns can still be represented implicitly. The use of such implicit representations reduces the cost of operations, therefore, it is useful to further generalize our extended X-relations. The implicitly universal variables are called "hidden" variables, as they are hidden in the tuples. The other variables are called "explicit" variables. This representation was first proposed in [56], and here we extend that representation by formalizing it in such a way that any kind of tables can be equipped with hidden variables.

Hidden table R_X^H representing an extended X-relation $\mathcal{R}$ has three components:

1. $T(R_X^H)$ denotes the internal table (which can be any kind of table);
2. $E(R_X^H)$ denotes the set of explicit variables;
3. $H(R_X^H)$ denotes the set of hidden variables.

Based on our definition of table data structures, any table represents an extended relation. The configuration of table with hidden variables R_X^H is such that:

1. $T(R_X^H)$ represents an extended $E(R_X^H)$-relation;
2. $E(R_X^H)$ and $H(R_X^H)$ are disjoint sets of variables;
3. $E(R_X^H) \cup H(R_X^H) = X$.

Function $\delta_{R_X^H}$, for this representation, is computed as:

$$\delta_{R_X^H}(\gamma) = \delta_{T(R_X^H)}(\gamma|_{E(R_X^H)})[\gamma].$$

Here, we abused the notation. By $T(R_X^H)$ in $\delta_{T(R_X^H)}$, we are referring the extended $E(R_X^H)$-relation that is represented by $T(R_X^H)$.

Before we describe how the relational algebraic operators are implemented for this representation, we need to introduce a new operation for tables with hidden variables. Let R_X^H be a table with hidden variables and S a subset of its hidden variables, i.e., $S \subseteq H(R_X^H)$. We define the multiplication operator to operate on a table with hidden variables and a set of variables, and produce a table with hidden variables, $T_X^H = R_X^H \times S$, such that

1. $H(T_X^H) = H(R_X^H) \backslash S$,
2. $E(T_X^H) = E(R_X^H) \cup S$,
3. For all $\gamma : X \mapsto A$, we have $\delta_T(\gamma) = \delta_R(\gamma)$.

Intuitively, the result of applying the multiplication operator on table R^H_X and set of variables S is a table with the same number of rows as R^H_X. The number of columns in the resulting table equals to the number of columns in R^H_X plus $|S|$.

Construction 4 *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *Let $\mathcal{T}$ be an extended X-relation defined as $\neg\mathcal{R}$. The following is used to construct a table with hidden variables for $\mathcal{T}$.*
   (a) $H(T^H_X) = H(R^H_X)$;
   (b) $E(T^H_X) = E(R^H_X)$;
   (c) $T(T^H_X) = \neg T(R^H_X)$.
2. *Let $\mathcal{T}$ be an extended $X \cup Y$-relation defined as $\mathcal{R} \bowtie \mathcal{S}$. The following is used to construct a table with hidden variables for $\mathcal{T}$.*
   (a) $H(T^H_{X\cup Y}) = \big(H(R^H_X) \cup H(S^H_X)\big) \setminus \big(E(R^H_X) \cup E(S^H_Y)\big)$;
   (b) $E(T^H_{X\cup Y}) = E(R^H_X) \cup E(S^H_X)$;
   (c) $T(T^H_{X\cup Y}) = T(R^H_X) \times \big(H(R^H_X) \cap E(S^H_Y)\big) \bowtie T(S^H_Y) \times \big(H(S^H_Y) \cap E(R^H_X)\big)$.
3. *Let $\mathcal{T}$ be an extended Z-relation defined as $\pi_Z(\mathcal{R})$. The following is used to construct a table with hidden variables for $\mathcal{T}$.*
   (a) $H(T^H_Z) = H(R^H_X) \cap Z$;
   (b) $E(T^H_Z) = E(R^H_X) \cap Z = Z \backslash H(R^H_X)$;
   (c) *Let S^H_X be the table with hidden variables defined as $T(R^H_X) \times \big(H(T^H_X) \backslash Z\big)$. Then, $T(T^H_Z) = \pi_{Z\backslash H(T^H_X)}\big(T(S^H_X)\big)$.*

Proposition 10 *Construction 4 is correct, that is, for each operator, the construction generates a table with hidden variables whose δ function satisfies the properties stated in Definition 16.*

Proof: Here, we provide the proofs for the negation and join operators.

- Correctness of construction for negation operator: By Definition 17, it is enough to show that $\delta_{T^H_X}(\gamma)$ and $\neg\delta_{R^H_X}(\gamma)$ are equivalent with respect to $\mathcal{A}$, for all γ.
  Let $\delta_{R^H_X}(\gamma) = \phi$. In construction, we define $T(T^H_X) = \neg T(R^H_X)$, and, based on the definition of the extended relation, formulas $\delta_{T(T^H_X)}(\gamma)$ and $\neg\delta_{T(R^H_X)}$ are equivalent with respect to $\mathcal{A}$. Since the set of hidden variables and explicit variables of the two tables, T and R, are the same, $\delta_{T^H_X}(\gamma) = \neg\phi = \neg\delta_{R^H_X}(\gamma)$.
- Correctness of construction for join operator: By Definition 17, it is enough to show $\delta_{T^H_X}(\gamma)$ and $\delta_{R^H_X}(\gamma) \wedge \delta_{S^H_X}(\gamma)$ are equivalent with respect to $\mathcal{A}$.
  Let $\gamma : X \cup Y \mapsto A$, $\gamma_1 = \gamma|_{X\cup\left(H(R^H_X)\cap E(S^H_Y)\right)}$ and $\gamma_2 = \gamma|_{Y\cup\left(H(S^H_Y)\cap E(R^H_X)\right)}$.
  We define $U^H_{X\cup\left(H(R^H_X)\cap E(S^H_Y)\right)}$ as the result of multiplication of table $T(R^H_X)$ and set of variables $H(R^H_X) \cap E(S^H_Y)$. According to the definition of multiplication:

$$\delta_{U^H_{X\cup\left(H(R^H_X)\cap E(S^H_Y)\right)}}(\gamma_1) = \delta_{T(R^H_X)}(\gamma_1|_X)[\gamma_1] = \delta_{R^H_X}(\gamma_1|_X) = \delta_{R^H_X}(\gamma|_X).$$

We define $V^H_{Y\cup\left(H(S^H_Y)\cap E(R^H_X)\right)}$ as the result of the multiplication of table $T(S^H_Y)$ and set of variables $H(S^H_Y)\cap E(R^H_X)$. Like the previous step, we can show

$$\delta_{V^H_{Y\cup\left(H(S^H_Y)\cap E(R^H_X)\right)}}(\gamma_2)=\delta_{S^H_Y}(\gamma|_Y).$$

The construction defines $T(T^H_{X\cup Y})$ as the result of joining $U^H_{X\cup\left(H(R^H_X)\cap E(S^H_Y)\right)}$ and $V^H_{Y\cup\left(H(S^H_Y)\cap E(R^H_X)\right)}$, so $\delta_{T(T^H_{X\cup Y})}(\gamma)[\gamma]=\delta_{T^H_{X\cup T}}(\gamma)$ is equivalent to $\delta_{R^H_X}(\gamma|_X)\wedge\delta_{S^H_Y}(\gamma|_Y)$, with respect to $\mathcal{A}$.

■

Tables with hidden variables represent certain extended relations compactly, e.g., the extended relation corresponding to the answer to an expansion predicate, or the extend relation corresponding to a tautology.

Example 13 *(Continuation of Example 10) Tables 5.16,...,5.20 represent answers in the form of Table with hidden variables whose inner tables are True/False tables, for $\phi_1,\ldots,\phi_5$, in the setting of Example 10.*

T:		
x	y	$\psi(x,y)$
2	1	$\top$
2	2	$\top$
2	3	$\top$
3	1	$\top$
$D=\bot$		
$H=\emptyset$		
$E=\{x,y\}$		

Table 5.16: Table with hidden variables for $\phi_1(x,y)$.

T:
$\psi(x,y)$
$Q(x,y)$
$D=\bot$
$H=\{x,y\}$
$E=\emptyset$

Table 5.17: Table with hidden variables for $\phi_2(x,y)$.

T:		
x	y	$\psi(x,y)$
2	1	$\bot$
2	2	$\bot$
2	3	$\bot$
3	1	$\bot$
$D=\top$		
$H=\emptyset$		
$E=\{x,y\}$		

Table 5.18: Table with hidden variables for $\phi_3(x,y)$.

T:		
y	z	$\psi(x,y,z)$
1	1	$Q(x,y,z)$
1	2	$Q(x,y,z)$
1	3	$Q(x,y,z)$
3	2	$Q(x,y,z)$
3	3	$Q(x,y,z)$
$D = \bot$		
$H = \{x\}$		
$E = \{y,z\}$		

Table 5.19: Table with hidden variables for $\phi_4(x,y,z)$.

T:	
y	$\psi(y)$
1	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
3	$Q(1,y) \vee Q(2,y) \vee Q(3,y)$
$D = \bot$	
$H = \{\}$	
$E = \{y\}$	

Table 5.20: Table with hidden variables for $\phi_5(y)$.

Comparing Table 5.17 and Table 5.7 illustrates the efficiency of using tables with hidden variables.

5.3.5 Tables with Restriction

We call a formula in the form x_1 *op* x_2, where *op* is a comparison operator, an *ordering formula*. Ordering formulas are used in specification to enforce a relation between variables and terms. In some problems, there are many tuples satisfying each individual ordering formula within the problem specification. However, the answer to the conjunction of such ordering formulas has many rows with false formula attached to them. As an example, we can mention the specification presented in Example 14.

Example 14 *In the blocked N queen problem, we have an N by N checkboard and N queens. Each square on the board can hold, at most one queen. Some squares are already blocked and cannot hold any queens. It is desired to place each of the N queens on to a non-blocked square such that no pair of queens can capture each other, i.e., no two queens are on the same row, column or diagonal. Given predicate Blk, representing the blocked cells, we use the binary predicate $Q(.,.)$ to represent the position of queens. Here, we restate specification BQ-03, presented in Section 3.3:*

$$\forall r, c \quad Blk(r,c) \rightarrow \neg Q(r,c) \tag{5.1}$$

$$\forall r_1, r_2, c \quad \neg Blk(r_1,c) \wedge \neg Blk(r_2,c) \wedge r_1 < r_2 \rightarrow \neg(Q(r_1,c) \wedge Q(r_2,c)) \tag{5.2}$$

$$\forall r, c_1, c_2 \quad \neg Blk(r,c_1) \wedge \neg Blk(r,c_2) \wedge c_1 < c_2 \rightarrow \neg(Q(r,c_1) \wedge Q(r,c_2)) \tag{5.3}$$

$$\forall r_1, r_2, c_1, c_2 \quad ((\neg Blk(r_1,c_1) \wedge \neg Blk(r_2,c_2)) \wedge |r_1 - r_2| = |c_1 - c_2|) \rightarrow \neg(Q(r_1,c_1) \wedge Q(r_2,c_2)). \tag{5.4}$$

The extended relation corresponding to an answer to atomic formula $|r_1 - r_2| = |c_1 - c_2|$ in (5.4), when the domain of each of r_1, r_2, c_1, c_2 is $\{1, \ldots, n\}$, has n^4 tuples (one tuple for each instantiation of the four variables). Among these n^4 tuples, $\theta(n^3)$ of them are mapped to *true*

formula, while $\theta(n^4)$ of them are mapped to *false* formula. So, neither a True-table nor a False-table represents this relation compactly. Note that there is no universal column (variable) in this table, and so we cannot use a table with hidden variables to represent this extended relation.

An interesting observation about axiom (5.4) is that we are joining the extended relation obtained for $|r_1 - r_2| = |c_1 - c_2|$, with the one computed for $\neg Blk(r_1, c_1) \wedge \neg Blk(r_2, c_2)$. If Blk is dense, i.e., the problem instance is not trivial, there are few rows whose formulas are *true* in the extended relation corresponding to the answer to $\neg Blk(r_1, c_1) \wedge \neg Blk(r_2, c_2)$. Therefore, the extended relation corresponding to the answer to $(\neg Blk(r_1, c_1) \wedge \neg Blk(r_2, c_2)) \wedge |r_1 - r_2| = |c_1 - c_2|$ has few rows which are mapped to *true*, and hence, it can be represented compactly using a False-table.

Using True/False tables, Enfragmo spends a lot of time computing and storing the tuples satisfying $|r_1 - r_2| = |c_1 - c_2|$, while many of those tuples are going to be eliminated when Enfragmo computes the answer to $|r_1 - r_2| = |c_1 - c_2| \wedge \neg Blk(r_1, c_1) \wedge \neg Blk(r_2, c_2)$. To avoid this extra computation, we propose using a new representation for extended relations, called "Table with Restriction".

Table with restriction R^r_X for extended X-relation $\mathcal{R}$ is composed of a True/False table, $T(R^r_X)$, which represents an extended Y-relation $I(R^r_X)$ where $Y \subseteq X$, and a formula (restriction) $F(R^r_X)$. The corresponding formula for assignment γ is the conjunction of the evaluation of the restriction formula under assignment γ, $F(R^r_X)[\gamma]$, and the formula which is mapped to $\gamma|_Y$ by extended Y-relation $T(R^r_X)$, i.e.,

$$\delta_{\mathcal{R}}(\gamma) = (F(R^r_X) \wedge \delta_{I(R^r_X)}(\gamma))[\gamma].$$

For example, consider atomic formula $\phi(\bar{x}) = P(\bar{x})$. The extended X-relation $\mathcal{R}$ can be described as a table with restriction R^r_X where $I(R^r_X)$ is an extended $\emptyset$-relation representing an answer to $\top$ and $F(R^r_X) = P(\bar{x})$.

Construction 5 *Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *Let $\mathcal{T}$ be an extended X-relation defined as $\neg\mathcal{R}$. The following is used to construct a table with restriction for $\mathcal{T}$.*
   (a) *$T(T^r_X)$ is the True/False table computed as $\neg(T(R^r_X) \bowtie F^{TF}_Y)$, where F^{TF}_Y is the True/False table representing an answer to formula $F(R^r_X)$,*
   (b) *$F(T^r_X)$ is simply $\top$ formula.*
2. *Let $\mathcal{T}$ be an extended $X \cup Y$-relation defined as $\mathcal{R} \bowtie \mathcal{S}$. The following is used to construct a table with restriction for $\mathcal{T}$.*
   (a) *$T(T^r_{X\cup Y})$ is the True/False table computed as $T(R^r_X) \bowtie T(S^r_Y)$,*
   (b) *$F(T^r_{X\cup Y})$ is the formula $F(R^r_X) \wedge F(S^r_Y)$.*
3. *Let $\mathcal{T}$ be an extended Z-relation defined as $\pi_Z(\mathcal{R})$. The following is used to construct a table with restriction for $\mathcal{T}$.*
   (a) *$T(T^r_Z)$ is the True/False table computed as $\pi_Z(T(R^r_X) \bowtie F^{TF}_Y)$, where F^{TF}_Y is the True/False table representing an answer to formula $F(R^r_X)$,*
   (b) *$F(T^r_Z)$ is formula $\top$.*

Proposition 11 *Construction 5 is correct, that is, for each operator it generates a table with restriction whose δ function satisfies the properties stated in Definition 16.*

Proof: Here, we provide the proof for the negation operator.

When we negate a table with restriction, we evaluate the restriction formula, $F(R^r_X)$, and so the formulas $\delta_{T^r_X}(\gamma)$ and $\neg\delta_{R^r_X}(\gamma)$ may not be equivalent formulas, but, as we show here, they are equivalent with respect to $\mathcal{A}$.

Let F^{TF}_Y be the True/False table representing an answer to formula $F(R^r_X)$. Since the restriction in table T^r_X is $\top$, by the definition of a table with restriction:

$$\delta_{T^r_X}(\gamma) \Leftrightarrow \delta_{T(T^r_X)}(\gamma).$$

Since F^{TF}_Y is the True/False table representing an answer to $F(R^r_X)$, we know that $\delta_{F^{TF}_Y}(\gamma)$ and $F(R^r_X)[\gamma]$ are equivalent with respect to $\mathcal{A}$.

We consider the following two cases:

1. $F(R^r_X)[\gamma]$ is evaluated as $\top$, under structure $\mathcal{A}$: formula $\delta_{R^r_X}(\gamma)$ is equivalent to $\delta_{T(R^r_X)}(\gamma)$ with respect to $\mathcal{A}$. Based on the construction, we set $\delta_{T(T^R_X)}(\gamma)$ to be $\neg\delta_{T(R^r_X)}(\gamma) \wedge \delta_{F^{TF}}(\gamma)$. Since $F(R^r_X)[\gamma]$ and $\delta_{F^{TF}}(\gamma)$ are equivalent with respect to $\mathcal{A}$, we conclude that $\delta_{T(T^R_X)}(\gamma)$ is equal to $\neg\delta_{T(R^r_X)}(\gamma)$ which is equivalent to $\neg\delta_{R^r_X}$ with respect to $\mathcal{A}$.

2. $F(R^r_X)[\gamma]$ is evaluated as $\bot$, with respect to structure $\mathcal{A}$: We have $\delta_{R^r_X}(\gamma)$ is equivalent to $\bot$. Based on the construction, we know $\delta_{T^r_X}(\gamma) = \delta_{T(R^r_X)}(\gamma) \vee \top \Leftrightarrow \top$. So, we conclude that $\delta_{T^r_X}(\gamma)$ and $\neg\delta_{R^r_X}(\gamma)$ are equivalent with respect to $\mathcal{A}$.

■

5.4 Algorithms

In this section, we briefly describe how an input to Enfragmo is encoded and also discuss the data structures and algorithms used in Enfragmo, to implement operations on tables.

As explained in Chapter 3, Enfragmo accepts two inputs; a problem specification and a problem instance. Both inputs must be in ASCII format and be consistent with the grammar presented in Appendix E. We already discussed the different parts of problem specifications, in Chapter 3. A problem instance has the following parts:

1. Interpretation for types,
2. Interpretation for instance predicates,
3. Interpretation for instance functions.

The interpretation for a type is encoded by listing all the elements in that type. The following illustrates how an enumerable type, Colour, and an integer type, LessThan5, are encoded for Enfragmo:

```
TYPE Colour {Red,Green,Blue}
TYPE LessThan5 {1,2,3,4,5}
```

To make presenting a problem instance easier, Enfragmo allows users to represent the integer types using a *compact representation*, which is then translated back to the above representation. Users can write the following to describe type $LessThan5$.

```
TYPE LessThan5 1..5
```

Remember that Enfragmo does not work with the compact representation of integer types, and the *compact representation* is transformed to the normal representation of types, by a preprocessor module.

The interpretations for instance predicates and functions are represented by simply listing the tuples in the interpretations.

Remark 1 *Let us assume a problem instance has a type with n distinct elements. Any encoding for representing n distinct elements needs $\theta(n \log n)$ bits. The size of a problem instance for this problem is $\Omega(n \log n)$, which is also $\Omega(n)$.*

The data structure used to represent an assignment to X is an array of size $|X|$. Enfragmo assumes an ordering on each of the given sorts. It uses the standard ordering for the Integer sorts and applies an arbitrary ordering for the enumerative sorts.

A ground formula is represented using a DAG (Directed Acyclic Graph) data structure. Each node of the DAG corresponds to a Boolean operator, e.g., AND, OR, NOT. In addition, there are special nodes, called assignment nodes, that map a value to a variable. This representation of formulas allows the reuse of some parts of a DAG formula in another one. For example, the formula $\Phi \wedge \Psi$ can be constructed by just creating a new AND node whose two children are the roots of the DAGs representing Φ and Ψ.

A row is a pair consisting of an assignment and a formula. The sets of rows in Basic, Normal, True/False tables are implemented as an array of rows.

In this section, we describe algorithms for computing the results of negation/join/projection operators on True/False tables. The algorithms for computing the results of negation/join and projection on Basic and Normal tables are very similar to their corresponding algorithms for True/False tables.

In Mohebali's Msc thesis [56], these operations were implemented using hash functions. That approach has the following drawback: There is no guarantee for the optimality of the selected hash function, and so the worst-case running time of a hash-based algorithm can be much worse than its average-case running time. Shahab Tasharrofi and the author designed algorithms whose worst-case running times are the same as, within a constant factor, the average-case running times of the algorithms proposed in [56].

5.4.1 Relational Algebra Operations for True/False Table by Sorting

Let R_X^{TF} be a True/False table, $r_1 = (\gamma_1, \psi_1)$ and $r_2 = (\gamma_2, \psi_2)$ two rows of $T(R_X^{TF})$. We say r_1 is smaller than r_2 based on variable ordering $\langle X_1, \cdots, X_m \rangle$, if there exists an index i such that $\gamma_1|_{X_i} \leq \gamma_2|_{X_i}$ and for every $j < i$, $\gamma_1|_{X_j} = \gamma_2|_{X_j}$.

To sort a table, one can use any standard sorting algorithm, e.g., quick sort or heapsort [25]. The running time of these standard sorting algorithms on basic table T_X^B is $\theta(n^2|A|^n \log |A|)$, where

$n = |X|$, since comparing two rows costs $O(n)$, and there are $\theta(|A|^n \log |A|^n)$ comparisons to be made. Quick sort algorithm sorts True/False table T_X^{TF} in time $\theta(n|T(T_X^{TF})| \log |T(T_X^{TF})|)$. The author of this thesis designed an algorithm based on the idea used in radix-sort [25], that sorts a Basic table in $n|A|^n$ and a True/False table in $n|T(T_X^{TF})|$. The details of this algorithm can be found in Appendix F. We compare the performance of this sorting algorithm and the quicksort algorithm in Section 5.5.

Negation

To negate True/False table R_X^{TF}, we simply scan the rows in $T(R_X^{TF})$ and negate the formula part of each row. We must set the default formula of the resulting table as $\neg D(R_X^{TF})$.

Algorithm 2 Algorithm for Negating a True/False Table.

Input: R_X^{TF} is a True/False table
Output: S_X^{TF}, is a True/False table representing $\neg R_X^{TF}$
1: **function** NEGATE(R_X^{TF})
2: **for** $\langle \gamma, \phi \rangle \in T(R_X^{TF})$ **do**
3: $T(S_X^{TF}) = T(S_X^{TF}) \cup \{\langle \gamma, \neg\phi \rangle\}$
4: Set $D(S_X^{TF}) = \neg D(R_X^{TF})$.

Proposition 12 *Algorithm 2 returns True/False table S_X^{TF} that satisfies the properties described in Construction 3. The running time of this algorithm is $\theta(|nT(R_X^{TF}|)$, where $n = |X|$.*

Proof: Algorithm 2 visits every pair in $T(R_X^{TF})$ exactly once and sets the content of $T(S^{TF})$ as defined in Construction 3. Since there are $|T(R_X^{TF})|$ pairs in $T(R_X^{TF})$ and we need to copy each pair, we conclude that the running time of this algorithm is $\theta(n|T(R_X^{TF}|)$. ■

Join

As we discussed in Subsection 5.3.3, there are three different cases for joining two True/False tables:

1. **Joining Two False Tables:**

   The result of joining False table R_X^{TF} and False table S_Y^{TF} is False table T_Z^{TF} where $Z = X \cup Y$. For each (γ_1, ϕ_1) in $T(R_X^{TF})$ and (γ_2, ϕ_2) in $T(S_Y^{TF})$ such that $\gamma_1|_{X \cap Y}$ and $\gamma_2|_{X \cap Y}$ are the same assignment, we insert $\langle \gamma, \phi_1 \wedge \phi_2 \rangle$ into $T(T_Z^{TF})$ where γ is an assignment from $Z \mapsto A$, $\gamma|_X = \gamma_1$ and $\gamma|_Y = \gamma_2$. Having tables R_X^{TF} and S_Y^{TF} sorted based on the appropriate variable ordering, table T_Z^{TF} can be produced in linear time with respect to the number of elements in the input and output tables.

Algorithm 3 Algorithm for Joining two False Tables.

Input: False tables R_X^{TF} and S_Y^{TF}
Output: False table $T_Z^{TF} = R_X^{TF} \bowtie S_Y^{TF}$

1: $Z = X \cup Y, C = X \cap Y, T_Z^{TF} =$ empty array ;
2: Sort $T(R_X^{TF})$ and $T(S_Y^{TF})$ based on variable ordering O where O prefers variables in C to the other variables.
3: RRowInd = SRowInd = 0 These variables indicate the row of table R and T we are processing.
4: **while** $RRowInd < |T(_X^{TF})|$ **and** $SRowInd < |T(S_Y^{TF})|$ **do**
5: Let $\langle \gamma_1, \phi_1 \rangle$ ($\langle \gamma_2, \phi_2 \rangle$) be row indexed RRowInd (SRowInd) in R_X^{TF} (S_Y^{TF}, respectively).
6: **if** $\gamma_1|_C < \gamma_2|_C$, based on variable ordering O **then**
7: RRowInd= RRowInd+ 1
8: **else if** $\gamma_1|_C > \gamma_2|_C$, based on variable ordering O **then**
9: SRowIndex= SRowIndex+ 1
10: **else**
11: RLast= index of the last row in $T(R_X^{TF})$ whose assignment agrees with γ_1 on C.
12: SLast= index of the last row in $T(S_Y^{TF})$ whose assignment agrees with γ_2 on C.
13: **for** each tuple $\gamma : Z \mapsto A$ s.t., $\langle \gamma|_X, \phi_1 \rangle$ ($\langle \gamma|_Y, \phi_2 \rangle$) is a row in $T(R_X^{TF})[RRowInd..RLast]$ ($T(S_Y^{TF})[SRowInd..RLast]$, respectively) **do**
14: Add $\langle \gamma, \phi_1 \wedge \phi_2 \rangle$ to $T(T_Z^{TF})$.
15: RRowInd= RLast+ 1
16: SRowInd= SLast+ 1

2. Joining a False table and a True table:

The result of joining False table R_X^{TF} and True table S_Y^{TF} is False table T_Z^{TF} where $Z = X \cup Y$. Joining a True table with a False table is similar to joining two False tables. We must consider the tuples which are not present in the True table. Algorithm 4 shows how we can join a True table with a False table.

Algorithm 4 Algorithm for Joining a False Table with a True Table.

Input: False table R_X^{TF} and True table S_Y^{TF}
Output: False table $T_Z^{TF} = R_X^{TF} \bowtie S_Y^{TF}$

$Z = X \cup Y, C = X \cap Y, T_Z^{TF} =$ empty array ;
Sort $T(R_X^{TF})$ and $T(S_Y^{TF})$ based on variable ordering O where O prefers variables in C to the other variables.
RRowInd = SRowInd = 0 These variables indicate the row of table R and T we are processing.;
while $RRowInd < |T(R_X^{TF})|$ **and** $SRowInd < |T(S_Y^{TF})|$ **do**
    Let $\langle \gamma_1, \phi_1 \rangle$ ($\langle \gamma_2, \phi_2 \rangle$) be row indexed RRowIndex (SRowIndex) in R_X^{TF} (S_Y^{TF}, respectively).
    if $\gamma_1|_C < \gamma_2|_C$, based on variable ordering O **then**
        for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_1|_C$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
            Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.
        RRowInd= RRowInd+ 1
    else if $\gamma_1|_C > \gamma_2|_C$, based on variable ordering O **then**
        SRowIndex= SRowIndex+ 1
    else
        RLast= index of the last row in $T(R_X^{TF})$ whose assignment agrees with γ_1 on C.
        SLast= index of the last row in $T(S_Y^{TF})$ whose assignment agrees with γ_2 on C.
        for each tuple $\gamma : Z \mapsto A$ s.t., $\langle \gamma|_X, \phi_1 \rangle$ ($\langle \gamma|_Y, \phi_2 \rangle$) is a row in $T(R_X^{TF})[RRowInd..RLast]$ ($T(S_Y^{TF})[SRowInd..RLast]$, respectively) **do**
            Add $\langle \gamma, \phi_1 \wedge \phi_2 \rangle$ to $T(T_Z^{TF})$.
        for each $\gamma : Z \mapsto A$ s.t., $\gamma|_X$ is in $T(R_X^{TF})[RRowInd..RLast]$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
            Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.
        RRowInd= RLast+ 1
        SRowInd= SLast+ 1
while $RRowInd < |T(R_X^{TF})|$ **do**
    Let $\langle \gamma_1, \phi_1 \rangle$ be row indexed RRowInd in R_X^{TF}.
    for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_1|_C$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
        Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.

3. **Joining two True Tables:** Joining two True tables is similar to joining two False tables except that we need to consider the tuples that are not in the True tables. The algorithm for joining two True tables is described in Algorithm 5.

Algorithm 5 Algorithm for Joining two True Tables.

Input: True tables R_X^{TF} and S_Y^{TF}
Output: True table $S_Z^{TF} = R_X^{TF} \bowtie S_Y^{TF}$

$Z = X \cup Y, C = X \cap Y, T_Z^{TF} =$ empty array ;
Sort $T(R_X^{TF})$ and $T(S_Y^{TF})$ based on variable ordering O such that O prefers variables in C to the other variables.
RRowInd = SRowInd = 0 These variables indicate the row of table R and T we are processing.;
while $RRowInd < |T(R_X^{TF})|$ **and** $SRowInd < |T(S_Y^{TF})|$ **do**
    Let $\langle \gamma_1, \phi_1 \rangle$ ($\langle \gamma_2, \phi_2 \rangle$) be row indexed RRowInd (SRowInd) in R_X^{TF} (S_Y^{TF}, respectively).
    if $\gamma_1|_C < \gamma_2|_C$, based on variable ordering O **then**
        for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_1|_C$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
            Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.
        RRowInd= RRowInd+ 1
    else if $\gamma_1|_C > \gamma_2|_C$, based on variable ordering O **then**
        for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_2|_C$ and there is no row with assignment $\gamma|_X$ in $T(R_X^{TF})$ **do**
            Add $\langle \gamma, \phi_2 \rangle$ to $T(T_Z^{TF})$.
        SRowIndex= SRowIndex+ 1
    else
        RLast= index of the last row in $T(R_X^{TF})$ whose assignment agrees with γ_1 on C.
        SLast= index of the last row in $T(S_Y^{TF})$ whose assignment agrees with γ_2 on C.
        for each tuple $\gamma : Z \mapsto A$ s.t., $\langle \gamma|_X, \phi_1 \rangle$ ($\langle \gamma|_Y, \phi_2 \rangle$) is a row in $T(R_X^{TF})[RRowInd..RLast]$ ($T(S_Y^{TF})[SRowInd..RLast]$, respectively) **do**
            Add $\langle \gamma, \phi_1 \wedge \phi_2 \rangle$ to $T(T_Z^{TF})$.
        for each $\gamma : Z \mapsto A$ s.t., $\langle \gamma|_X, \phi_1 \rangle$ is in $T(R_X^{TF})[RRowInd..RLast]$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
            Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.
        for each $\gamma : Z \mapsto A$ s.t., $\langle \gamma|_Y, \phi_2 \rangle$ is in $T(S_Y^{TF})[RRowInd..RLast]$ and there is no row with assignment $\gamma|_X$ in $T(R_X^{TF})$ **do**
            Add $\langle \gamma, \phi_2 \rangle$ to $T(T_Z^{TF})$.
        RRowInd= RLast+ 1
        SRowInd= SLast+ 1
while $RRowInd < |T(R_X^{TF})|$ **do**
    Let $\langle \gamma_1, \phi_1 \rangle$ be row indexed RRowInd in R_X^{TF}.
    for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_1|_C$ and there is no row with assignment $\gamma|_Y$ in $T(S_Y^{TF})$ **do**
        Add $\langle \gamma, \phi_1 \rangle$ to $T(T_Z^{TF})$.
while $SRowInd < |T(S_Y^{TF})|$ **do**
    Let $\langle \gamma_2, \phi_1 \rangle$ be row indexed SRowInd in S_Y^{TF}.
    for each $\gamma : Z \mapsto A$ s.t., $\gamma|_C = \gamma_2|_C$ and there is no row with assignment $\gamma|_X$ in $T(R_X^{TF})$ **do**
        Add $\langle \gamma, \phi_2 \rangle$ to $T(T_Z^{TF})$.

In Algorithms 3, 4 and 5, it is necessary to find the last row which is consistent with a given assignment. We can use either a linear or binary search to find the index of that row.

Proposition 13 *Algorithms 3,4 and 5 return True/False table T_Z^{TF} that satisfies the properties described in Construction 3. The running times of these algorithms are $O(n \times \max(|T(R_X^{TF})| + |T(S_X^{TF})|, |T(T_Z^{TF})|))$, where $n = |X|$.*

Proof: Here, we only present the proof for the correctness of Algorithm 4.

Based on Construction 3, $T(T_Z^{TF})$ is the union of two sets:

- In line 8 and the second while-loop (lines 21-24), we add a pair to $T(T_X^{TF})$ which corresponds to a member of set $\{(\gamma, \psi) \mid (\gamma|_X, \psi) \in T(R_X^{TF})$ and $\gamma|_Y \notin T(S_Y^{TF})\}$.
- In line 15 to 20, we add pairs to $T(T_X^{TF})$ corresponding to members of set $\{(\gamma, \psi) \mid \gamma|_X \in T(R_X^{TF})$ and $\gamma|_Y \in T(S_Y^{TF})$ and $\delta_{\mathcal{S}}(\gamma|_Y) \neq \bot, \psi = \delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y)\}$.

In each iteration, we add one element to $T(T_Z^{TF})$ and, since each row has a tuple of length n, the running time of this algorithm is at most $n|T_Z^{TF}|$. On the other hand, in each iteration, we increase either $RRowInd$ or $SRowInd$ or sometimes both, so the number of iterations can not be more than $|S_X^{TF}| + |R_Y^{TF}|$.

■

Projection

The result of projecting False table R_X^{TF} on variables Z is False table T_Z^{TF}. We need to collect all the rows in $T(R_X^{TF})$ which have the same value on variables in Z and disjunct the formula part of those rows. For True tables, if at least one of the collected rows is mapped to True formula, the formula corresponding to the assignment in T_Z^{TF} is True formula, and hence, no row should be inserted in the resulting table. Algorithm 6 represents an algorithm for the projection operation on True/False tables.

Algorithm 6 Algorithm for Projecting a True/False Table.

Input: True/False table R_X^{TF}, and a set of variables $D \subseteq X$
Output: True/False table $T_Z^{TF} = \pi_Z(R)$

1: Sort $T(R_X^{TF})$ based on variable ordering $\langle Z_1, \cdots, Z_{|X|}, \cdots \rangle$, where $Z = \{X_1, \cdots, X_{|Z|}\}$.
2: **for** each assignment γ of the set of variables $X \setminus Z$ **do**
3: Let R be the set of tuples in $T(R_X^{TF})$ sharing the same values for the variables $X \setminus Z$ as γ
4: Let $\Psi = \bigvee_{\langle \gamma, \phi \rangle \in R} \phi[\gamma|_Z]$
5: **if** $D(R^{TF}) = \bot$ **or** $D(R^{TF}) = \top$ and R contains all possible instantiations of variables in $X \setminus Z$ **then**
6: $T(T_Z^{TF}) = T(T_Z^{TF}) \cup \{\langle \gamma|_Z, \Psi \rangle\}$

The set of tuples sharing the same value for a set of variables in Algorithm 6 can be constructed either by using a linear or binary search.

Proposition 14 *Algorithm 6 returns True/False table T_Z^{TF} which satisfies the properties described in Construction 3. Also, the running time of this algorithm is $O(n \times \max(|T(R_X^{TF})|, |D_{X \setminus Z}|))$, where $n = |X|$.*

Proof: Based on Construction 3, $T(T_Z^{TF}) = \{(\gamma, \psi) \mid \gamma$: there is at least one γ' s.t. $\gamma' \in T(R_X^{TF})$ and $\gamma = \gamma'|_Z, \psi = \bigvee_{\{\gamma': X \to A | \gamma = \gamma'|_Z\}} \delta_{\mathcal{R}}(\gamma')[\gamma'|_{(X \setminus Z)}]\}$. Line 3 collects all the pairs that share the same assignment to the variables in Z (this corresponds to $\gamma = \gamma'|_Z$). Line 4 constructs the disjunction of the corresponding formulas and line 6 adds the pair into T_Z^{TF}.

The for-loop, line 2, scans all the elements of $D_{X \setminus Z}$. Given that the elements in $T(R_X^{TF})$ are sorted, all the operations inside the for-loop can be implemented in constant time.

■

Complexity of Operations

Assuming the tables are sorted based on an appropriate variable ordering, Table 5.21 summarizes the worst-case complexity of performing the negation, join and projection operations.

Operation	Complexity
$T_{X \cup Y}^{TF} = R_X^{TF} \bowtie S_Y^{TF}$	$\theta(\lvert X \cup Y \rvert \times \max(\lvert T(R_X^{TF}) \rvert + \lvert T(S_Y^{TF}) \rvert, \lvert T(T_Z^{TF}) \rvert))$
$T_Z^{TF} = \pi_Z(R_X^{TF})$	$\lvert X \rvert \times \lvert T(R_X^{TF}) \rvert$
$T_X^{TF} = \neg(R_X^{TF})$	$\lvert X \rvert \times \lvert T(R_X^{TF}) \rvert$

Table 5.21: Worst-case Complexity of Performing Operations of True/False Tables.

The operations for the other tables, Basic tables, Normal tables and etc, can be implemented using either one of the above algorithms or a close variant of them.

Since Enfragmo uses the linear sorting algorithm described in Appendix F to sort the tables, the algorithms discussed in this section perform faster than those used in MXG [56], both from the worst-case and the average-case points of views.

5.5 Experimental Evaluation

In this section, we compare the grounding time of Enfragmo when it uses the described representations of the tables. We use the same encodings and benchmarks as the ones used in Section 3.4. Rows and columns in Table 5.22 correspond to specifications and representations, respectively. In Table 5.22, we compare the following four configurations:

- Normal: Normal tables without hidden variables;
- Normal+Hidden: Normal tables with support of hidden variables;
- TrueFalse: True/False tables with support of hidden variables;
- Restriction: Table with Restriction where the inner table is a True/False table with support of hidden variables.

Encoding	Normal	Normal + Hidden	TrueFalse	Restriction
SG-01	0.006/0.009	<0.001/<0.001	<0.001/<0.001	<0.001/<0.001
SG-01	0.006/0.009	<0.001/<0.001	<0.001/<0.001	<0.001/<0.001
SG-01	0.006/0.009	<0.001/<0.001	<0.001/<0.001	<0.001/<0.001
BQ-01	0.007/0.007	0.007/0.007	0.006/0.006	0.005/0.005
BQ-02	0.086/0.077	0.53/0.51	0.11/0.10	0.10/0.10
BQ-03	22.43/22.40	15.87/15.51	15.88/15.48	15.86/15.45
BQ-04	4.828/4.826	4.34/4.29	4.34/4.29	4.33/4.30
GC-01	0.157/0.159	0.144/0.143	0.139/0.137	0.138/0.137
GC-02	0.390/0.380	0.389/0.379	0.382/0.374	0.382/0.368
GC-03	0.363/0.350	0.327/0.316	0.324/0.309	0.321/0.309
GC-04	0.380/0.369	0.406/0.392	0.406/0.392	0.406/0.392
HP-01	6.720/6.189	6.720/6.189	4.511/4.291	4.508/4.287
HP-02	0.019/0.017	0.017/0.016	0.009/0.009	0.009/0.009
HP-03	*/*	*/*	*/*	*/*
HP-04	4.071/4.043	4.033/4.008	2.353/2.305	2.351/2.305

Table 5.22: Performance comparison of different tables based on grounding time. A row (column) represents a problem specification (a grounding configuration, respectively). There are two entries in each cell. The first (second) entry is the grounding time, in seconds, when Enfragmo uses Quicksort (RadixSort, respectively). None of the configurations are able to ground all the Hamiltonian path instances using specification HP-03 within 10 minutes of timeout.

Based on the data presented in Table 5.22, we observe the following:

1. Using Hidden variables speeds up the grounding process: This can be observed by comparing the first two columns of Table 5.22.
2. As we expected, the grounding time of configurations using Radix sort is smaller than their corresponding configurations using Quick sort.
3. In most cases, grounding using True/False Table performs better than using Normal tables.
4. When we use Quick sort as the sorting algorithm, Table with restrictions never performs worse than True/False tables.
5. We see a speed-up, in the grounding when we use table with restrictions in the Blocked N-Queens and the Graph Colouring problems, as there are ordering formulas in the specifications which are conjuncted with the instance predicates.

5.6 Conclusion

In this chapter, we described how Enfragmo handles the grounding phase, in which, given a specification and an instance, it generates a variable free first-order formula equivalent to the problem instance. The grounding algorithm presented in this chapter is an extension of work reported in [58]. We generalized their grounding algorithm in several ways. We presented our own grounding algorithm based on an abstract notion of tables, which support certain operations. We also generalized the notion of answer to formula, which enabled us to introduce table with restriction.

We proposed different representations for a table. True/False Tables and Table with restriction are novel data structures, and have not been proposed elsewhere. Generalizing the table with hidden variables is another novel idea presented in this chapter. In addition, for each representation we described a new algorithm for computing the results of extended relational algebraic operators. From our previous experiments we already know that different representations of tables have different performances. We used some experiments to show that our intuition about the performance of each representation is correct.

For computing the results of relational algebraic operators, we proposed sorting-based algorithms whose worst-case complexities are the same as the average-case complexities of hash-based algorithms. We believe one of the reasons for the good performance of Enfragmo is these fast algorithms.

Chapter 6

Grounding Specifications with Complex Terms

In the previous chapter, we describe a framework for grounding the specifications expressed in first-order logic. In this chapter, we describe how to extend the method, introduced in Chapter 5, such that it can ground the specifications expressed in the first-order logic extended with arithmetic, functions and aggregate operators, in polynomial time.

6.1 Introduction

As described in Chapter 3, a term in an Enfragmo specification is a variable, application of an instance/expansion function, application of an arithmetic operator on integer terms, or an aggregate applied to an appropriate number of formulas and terms. In Chapter 5, we described how Enfragmo deals with specifications whose terms are just variables. In this chapter, we explain how Enfragmo grounds specifications with non-variable terms.

Adding function, arithmetic and aggregate terms to the logic on which the model expansion task is defined can cause certain issues:

1. The expressive power of the logic may be increased in the presence of arithmetic and aggregate operators, and so polynomial time reduction to an NP-complete problem such as SAT may no longer be possible. Moreover, since the domain of arithmetic is infinite, finite grounding may not be possible.
2. By adding these new constructs to the logic, there is no guarantee for the terms to be total. For example, the value of term x/y is undefined when y takes the value 0.
3. Arithmetic and aggregate operators can enlarge the set of values (range) a term can take. For example, term $x \times y + z$ where all variables range over the domain $\{0, \cdots, 10\}$ can take all values in the set $\{0, \cdots, 110\}$. Having a large domain may cause existing grounding algorithms to work very slowly.

The first issue can be resolved by restricting the syntax and constructing a new fragment with a controlled expressive power. For example, Ternovska and and Mitchell showed that the fragment of double-guarded first-order logic captures NP (under "small-cost" condition) [65]. In another paper

by our group, Tasharrofi and Ternovska proposed a logic that allows users to represent NP problems involving arithmetic "naturally" (i.e., with built-in arithmetic as opposed to using binary encoding of integers) [64]. In their approach, NP is captured without any restriction.

There are several approaches to address the second issue. The most obvious approach is to restrict the syntax to avoid partially defined terms. Another approach is to use appropriate guards, e.g., to rewrite $P(x\backslash y)$ as $\exists z : x = z \times y \wedge P(z)$. One can also deal with this issue by making all functions total. The only terms which are not total in the input language of Enfragmo are Min and Max aggregates. Based on the system language, users must provide a default value for these aggregates. If the value of a Min/Max aggregate is undefined (based on the standard definitions of these aggregates), Enfragmo defines the value of the aggregate to be the "default" value [7].

The third issue is related to the complexity of grounding. Enriching the language may result in the poor performance of the previous grounding algorithms and may force us to introduce a new grounding approach. One way to avoid this issue is introduce new algorithms to handle the new constructs.

In the context of finite model expansion, arithmetic operators are not total, since the results of these operators may be an element which does not belong to the universe of the instance structure. For example, the result of the summation of two variables ranging over $D = \{1..5\}$ is not always in D. So, during the grounding process, we may encounter values which are not given in the instance structure $\mathcal{A}$. We have the same scenario for *aggregates*. To avoid this issue, we developed Enfragmo based on embedded model expansion. As described in Section 2.3, in embedded model expansion, besides the instance structure, there is an infinite arithmetic structure.

The input language of Enfragmo assumes that the instance file passed to it describes structure $\mathcal{A} = (U, R)$ where U is the set of all integers, and R is a set of finite relations. As we explained in Chapter 3, Enfragmo supports two different kinds of sorts (types): Enumerable and Integer sorts (types). Enfragmo maps each element of enumerable sorts to an integer in U, so that all the elements in the active domain are integers.

We already described a generic approach to ground a function-free first-order formula over a given finite domain. Expressing most interesting real-world problems, e.g., the Traveling Salesman or Knapsack problems, in function-free FO logic without having access to arithmetical operators is not an easy task. So enriching the syntax with functions and arithmetical operators is a necessity. In this chapter, we introduce an extension of the grounding techniques described in the previous chapter, such that they can be used in the presence of these constructs.

Since the terms can only occur as the arguments to the atomic formulas, in order to support new kinds of terms, we only need to extend the way our grounding framework handles the atomic formula, and do not need to change any other components of our grounding algorithm.

6.1.1 My Contributions

The initial relational algebraic grounding approach has been proposed for function-free logic [58, 56]. The author extended the grounding approach to work in the presence of functions, arithmetical terms and aggregates. Also the two notions of answer to terms and the idea of term tables are proposed by the author of this thesis [7]. The algorithms to compute term tables and all the grounding algorithms described in this chapter are designed by the author.

6.2 Background

The notation used in this chapter is the same as that introduced in Section 2.1 and Chapter 5.

6.2.1 FO MX with Arithmetic

The grounding algorithm presented in the previous chapter works for specifications whose terms are just variables while, as presented in Chapter 3, the input language of Enfragmo includes more complex terms. In this chapter, we are concerned with specifications written in FO logic which is extended with functions, arithmetic and aggregate operators. We assume that the domain of any instance structure is a subset of $\mathbb{N}$ (set of natural numbers), and that arithmetic operators have their standard meanings. Details of aggregate operators need to be specified, but these also behave according to our normal intuitions. The specification language of Enfragmo restricts the range of quantified variables and instance functions, as well as the possible interpretations of expansion predicates and functions to finite subsets.

As we already discussed in Chapter 3, Enfragmo allows each variable to have its own domain; however, for the sake of explanation, we assume that all variables range over type T, where its interpretation, T, is a subset $\mathbb{N}$. Notice that T and T are different. The former is a type declared in the specification while the latter is an interpretation for type T, defined in the problem instance.

Under the assumption that all the variables are associated with the same type T, Enfragmo rewrites $\phi(t_1(\bar{x}), \cdots, t_k(\bar{x}))$ as $\exists y_1, \cdots, y_k : y_1 = t_1(\bar{x}) \wedge \cdots \wedge y_k = t_k(\bar{x}) \wedge \phi(y_1, \cdots, y_k)$, where each y_i is associated with same type T. By applying this rewriting on the specifications, each atomic formula in the specifications is one of the following:

1. $P(x_1, \cdots, x_k)$, where P is an arity k predicate and x_i is a variable of type T;
2. $y = t(\bar{x})$, where t is a term and all variables are associated with type T;
3. $t_1(\bar{x})$ *op* $t_2(\bar{x})$, where t_1 and t_2 are terms, *op* is a comparison operator and all variables are associated with type T.

Since $t_1(\bar{x}) > t_2(\bar{x}) \Leftrightarrow t_2(\bar{x}) < t_1(\bar{x})$, $t_1(\bar{x}) \geq t_2(\bar{x}) \Leftrightarrow t_2(\bar{x}) \leq t_1(\bar{x})$ and $t_1(\bar{x}) \leq t_2(\bar{x}) \Leftrightarrow (t_1(\bar{x}) < t_2(\bar{x}) \vee t_1(\bar{x}) = t_2(\bar{x}))$, we can rewrite specifications such that they do not have any atomic formula in form $t_1(\bar{x})\{\leq, >, \geq\}t_2(\bar{x})$.

Syntax and Semantics of Aggregate Operators

In Chapter 3, we described the syntax and semantics of aggregates, but to make the content of this chapter self-contained, we present them here, again.

- $t(\bar{y}) = Max_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M(\bar{y})\}$, for any instantiation $\bar{b}$ for $\bar{y}$, denotes the maximum value obtained by $t(\bar{a}, \bar{b})$ over all instantiations $\bar{a}$ for $\bar{x}$ for which $\phi(\bar{a}, \bar{b})$ is *true*, or d_M if there is none.
- $t(\bar{y}) = Min_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_m(\bar{y})\}$ is defined dually to Max.
- $t(\bar{y}) = Sum_{\bar{x}}\{t(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y})\}$, for any instantiation $\bar{b}$ of $\bar{y}$, denotes 0 plus the sum over values of $t(\bar{a}, \bar{b})$, for all instantiation of $\bar{a}$ for $\bar{x}$, for which $\phi(\bar{a}, \bar{b})$ is *true*.
- $t(\bar{y}) = Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\}$, for any instantiation $\bar{b}$ for $\bar{y}$, denotes the number of tuples $\bar{a}$ for which $\phi(\bar{a}, \bar{b})$ is *true*, where $\bar{a}$ is an instantiation for $\bar{x}$.

Through out this chapter, $\mathcal{A}$ always denotes a finite σ-structure, called the instance structure, σ is the instance vocabulary, and ε the expansion vocabulary, $\mathcal{L}$ is the input language of Enfragmo,

defined in Chapter 3 and ϕ is the conjunction of axioms in a problem specification.

6.3 A Variant of the Knapsack Problem (Running Example)

In this section, we review Example 5 and use it as a running example throughout this chapter.

Example 15 *We are given a set of items (loads), $L = \{1, \cdots, n\}$, and the weight of each item is specified by an instance function W which maps items to integers ($w_i = W(i)$). We want to check if there is a way to put these n items into m knapsacks, $K = \{1, \cdots, m\}$, while satisfying the following constraints:*

Certain items must be placed into certain knapsacks. These pairs are specified using the instance predicate "P". There are m knapsacks; h of which have a high capacity, meaning they can carry a total load of H_{Cap}, while the capacity of the other knapsacks is L_{Cap}. We cannot put two items whose weights are very different in the same bag, i.e., the difference between the weights of the items in the same bag must be less than W_l. Constant values H_{Cap}, L_{Cap}, W_l and h are described as instance functions.

To encode this problem, we use a binary expansion predicate $Q(,)$, where $Q(l,k)$ represents item l placed in knapsack k. Let Φ be the conjunction of the following axioms (A_1 to A_5). Then, formula Φ encodes this problem:

$$A_1 : \forall l \; Count_k\{Q(l,k)\} = 1;$$

$$A_2 : \forall l, k \; P(l,k) \rightarrow Q(l,k);$$

$$A_3 : \forall k \; Sum_l\{W(l) : Q(l,k)\} \; \leq H_{Cap};$$

$$A_4 : Count_k\{Sum_l\{W(l) : Q(l,k)\} \; \geq L_{Cap}\} \leq h;$$

$$A_5 : \forall k, l_1, l_2 \; (Q(l_1,k) \wedge Q(l_2,k)) \rightarrow (W(l_1) - W(l_2) \leq W_l)\,.$$

Axiom A_1 ensures that every load is placed in exactly one knapsack. Axiom A_2 is to ensure we respect the pre-arranged items. Axioms A_3 and A_4 control the load of the knapsacks. Finally, axiom A_5 avoids the placing of two items whose weights are too far from each other in the same knapsack.

An instance of this problem is a structure for vocabulary $\sigma = \{P, W, W_l, H_{Cap}, L_{Cap}, h\}$. The task is to find an expansion $\mathcal{B}$ of $\mathcal{A}$ that satisfies ϕ:

$$\underbrace{(L \cup K; P^{\mathcal{A}}, W^{\mathcal{A}}, W_l^{\mathcal{A}}, H_{Cap}^{\mathcal{A}}, L_{Cap}^{\mathcal{A}}, \overbrace{h^{\mathcal{A}}}^{\mathcal{A}}, \; Q^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

Interpretations of expansion vocabulary $\varepsilon = \{Q\}$ in structure $\mathcal{B}$ give us a mapping from items to knapsacks, satisfying the problem constraints.

Let's assume the following is an instance of this problem:

- $L = \{1, 2, 3\}$ *and* $K = \{1, 2\}$*;*
- $P^{\mathcal{A}} = \{(1, 1)\}$*;*
- $W^{\mathcal{A}} = \{1 \mapsto 2; 2 \mapsto 7; 3 \mapsto 4\}$*;*
- ${W_l}^{\mathcal{A}} = 4$*;*

- ${H_{Cap}}^{\mathcal{A}} = 8$;
- ${L_{Cap}}^{\mathcal{A}} = 6$;
- $h^{\mathcal{A}} = 1$.

6.4 Evaluating Arithmetic and Instance Functions

The relational algebra-based grounding algorithm, described in Chapter 5, is designed for function-free logics. In this and the following two sections, we extend this grounding algorithm so that it can handle specifications in which the terms have more complex forms. In this section, we present a simple method for special cases where terms do not contain expansion predicates/functions, so they can be evaluated purely based on the instance structure. We call these terms, *evaluable terms*.

Recall that an answer to subformula $\phi(\bar{x})$ is an extended X-relation $\mathcal{R}$, where $X = \{x \mid x \text{ occurs in } \bar{x}\}$. The tuples of extended relation $\mathcal{R}$ have length $|X|$. Now, consider an atomic formula, ψ, whose arguments are terms containing instance functions and arithmetic operators, e.g., $P(x + y)$. As discussed previously, Enfragmo rewrites ψ as $\exists z(z = x + y \wedge P(z))$. Although we have not yet discussed the handling of the sub-formula $z = x + y$, it is apparent that the answer to ψ, is an extended $\{x, y\}$-relation.

To modify the previous grounding algorithm, we add the following two cases to the base cases.

Definition 19 (Base Cases for Atomic Formulas with Evaluable Terms) *Let $\phi(\bar{x})$ be an atomic formula, and let t_1 and t_2 be terms used as arguments of ϕ. Extended X-relation $\mathcal{R}$, where $X = \{x \mid x \text{ occurs in } \bar{x}\}$, is an answer to ϕ with respect to $\mathcal{A}$, iff*

1. *If $\phi(\bar{x})$ is $t_1(\bar{x})$ op $t_2(\bar{x})$, where op $\in \{=, <\}$, then for every γ,*
   (a) *$\delta_{\mathcal{R}}(\gamma) = \top$ iff $\mathcal{A} \models t_1[\gamma]$ op $t_2[\gamma]$.*
   (b) *$\delta_{\mathcal{R}}(\gamma) = \bot$ iff $\mathcal{A} \not\models t_1[\gamma]$ op $t_2[\gamma]$.*
2. *If $\phi(\bar{x})$ is $x_1 = t_1(x_2, \cdots, x_n)$, then for every γ,*
   (a) *$\delta_{\mathcal{R}}(\gamma) = \top$ iff $\mathcal{A} \models (x_1 = t_1)[\gamma]$.*
   (b) *$\delta_{\mathcal{R}}(\gamma) = \bot$ iff $\mathcal{A} \not\models (x_1 = t_1)[\gamma]$.*

Example 16 *Let's consider the atomic formula $\phi(x, y) = W(x) - W(y) \leq W_l$, in Axiom A_5 of Example 15. Formula ϕ is an atomic formula of form $t_1 \leq t_2$, where $t_1(x, y)$ is $W(x) - W(y)$ and t_2 is W_l. Since both W and W_l are instance functions, we can use Definition 19 to compute the answer to atomic formula ϕ. Let γ be an assignment mapping x to 1 and y to 2, then we have:*

1. *$t_1^{\mathcal{A}}[\gamma] = W^{\mathcal{A}}(1) - W^{\mathcal{A}}(2) = 2 - 7 = -5$;*
2. *$t_2^{\mathcal{A}}[\gamma] = 4$.*

So, $\delta_{\mathcal{R}}(\gamma) = \top$.

Table 6.1 depicts a True/False table representing an answer to $\phi(x, y)$.

x	y	ψ
1	1	$\top$
1	2	$\top$
1	3	$\top$
2	2	$\top$
2	3	$\top$
3	1	$\top$
3	2	$\top$
3	3	$\top$
$D = \bot$		

Table 6.1: True/False table for an answer to $W(x) - W(y) \leq W_l$.

6.5 Answers to Terms - Part 1

Terms involving expansion functions and predicates, including aggregate terms whose vocabulary intersects with the expansion vocabulary, can only be evaluated with respect to a particular interpretation of those expansion predicates/functions. Thus, they cannot be evaluated during grounding, as in Section 6.4. We call a term which cannot be evaluated based on the instance structure a *complex term*.

In this section, we further extend the relational algebra-based grounding method to handle atomic formulas which have complex terms as their arguments. The key idea in our approach is to introduce the notion of an *answer to a term*. We then extend our grounding algorithm to compute answers to atomic formulas from answers to the terms which are their arguments. The terms we allow here include arithmetic expressions, instance functions, expansion functions, and aggregate operators involving them. The axioms A_3 and A_4 in Example 15 have these kinds of terms.

Let's restrict the terms in the input language of Enfragmo to those that can be constructed from instance and expansion functions, arithmetic operations, Count, Min and Max aggregates, or recursive application of these constructs. We call this language, which does not include the Sum aggregate, $\mathcal{L}'$. In this section, we focus on grounding axiomatization expressed in $\mathcal{L}'$ and in the next section, we discuss grounding specifications that have occurrences of Sum aggregates.

As we said before, we assume the only type in the specifications is the integer type T. All the constructions and definitions presented here can be extended for the general case.

Definition 20 (Answer to Term t wrt $\mathcal{A}$) *We say that $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ is* an answer to term $t(\bar{x})$ *with respect to $\mathcal{A}$ if, for every $a \in \alpha_{\mathcal{R}}$, the extended X-relation $\beta_{\mathcal{R}}(a)$, where $X = \{x \mid x \text{ occurs in } \bar{x}\}$, is an answer to the formula $(t = a)$ with respect to $\mathcal{A}$, and for every $a \notin \alpha_{\mathcal{R}}$, there is no structure $\mathcal{B}$, expanding $\mathcal{A}$ and no assignment γ, mapping variables in X to elements in T, such that $t^{\mathcal{B}}[\gamma] = a$.*

Intuitively, $\alpha_{\mathcal{R}}$ is the set of all possible values term $t(\bar{x})$ may take. The formula obtained from $\delta_{\beta_{\mathcal{R}}(a)}(\gamma)$, where γ is an assignment to variables occurring in $\bar{x}$, describes the condition under which $t[\gamma]$ evaluates to a. We may use $\delta_{\mathcal{R}}(\gamma, n)$ as a shorthand for $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$.

Note that Definition 20 allows $\alpha_{\mathcal{R}}$ to be either the exact range of t or a superset of the range of t. If $\alpha_{\mathcal{R}}$ is a superset of the exact range of t, it means some of the extended X-relations defined by $\beta_{\mathcal{R}}$ map all assignments to *false*.

Example 17 *(Continuation of Example 15) Assume $\psi(x, y)$ is $W(l_1) - W(l_2) \leq W_l$. Let $r = W_l$, $s_i = l_i$, $t_i = W(s_i)$ ($i \in \{1, 2\}$), $r' = t_1 - t_2$ be the terms in ψ, and $\mathcal{R}$, $\mathcal{S}_i$, $\mathcal{T}_i$ ($i \in \{1, 2\}$), $\mathcal{R}'$ be answers to these terms, respectively. Then $\alpha_{\mathcal{R}} = \{4\}$, $\alpha_{\mathcal{S}_i} = \{1, 2, 3\}$, $\alpha_{\mathcal{T}_i} = \{2, 4, 7\}$ and $\alpha_{\mathcal{R}'} = \{-5, -3, -2, 0, 2, 3, 5\}$. Alternatively, we can set $\alpha_{\mathcal{T}_2} = \{2, 4, 5, 7\}$ and $\delta_{\mathcal{T}_2}(\gamma, 5) = \bot$, for every γ.*

Below, we describe the properties that are sufficient for an extended relation to constitute an answer to a term. Based on Definition 20, an answer to a term has two components; $\alpha_{\mathcal{R}}$, and $\beta_{\mathcal{R}}$. We first present an algorithm, Algorithm 7, to compute $\alpha_{\mathcal{R}}$, for terms in language $\mathcal{L}'$.

Let $\bar{x}$ be a tuple of variables of size k. We define $D_{\bar{x}}$ to be the set of all k-tuples of domain elements, i.e., $D_{\bar{x}} = T^k$.

Remark 2 *Let $t(\bar{x})$ be a term and $\mathcal{A}$ a σ-structure containing an interpretation for every function and predicate in t, i.e., $vocab(t) \subseteq \sigma$. The exact range of t, i.e., the values t takes under different assignments, can be computed by evaluating $t^{\mathcal{A}}[\gamma]$, for all possible assignments γ, mapping variables occurring in $\bar{x}$ to elements in T. If the vocabulary of t contains expansion predicates/functions, or in other words (that is $\mathcal{A}$ does not interpret all the functions and predicates in t), then Algorithm 7 computes a superset of the range of t, SuperRange of t, by finding all possible values t may take, under all expansions of $\mathcal{A}$ and all assignments.*

Given σ-structure $\mathcal{A}$, the exact range of term $t(\bar{x})$, where $vocab(t) \subseteq \sigma$, can be computed in linear time with respect to the size of $D_{\bar{x}}$. However, since Algorithm 7 computes its return value fast enough, we always use the return value of this algorithm as $\alpha_{\mathcal{R}}$.

Algorithm 7 Computing $SuperRange\ t(\bar{x})$

Input: t (a term in $\sigma \cup \varepsilon$), T (interpretation for type T), $\mathcal{A}$ (a σ-structure).
Output: $\alpha_{\mathcal{R}}$: $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ is an answer to t wrt to $\mathcal{A}$.

1: **function** COMPUTESUPERRANGE(t, T, $\mathcal{A}$)
2: **if** t is a variable **then**
3: **return** T
4: **else if** $t = t_1 + t_2$ **then**
5: $\alpha_1 = ComputeSuperRange(t_1, T, \mathcal{A}), \alpha_2 = ComputeSuperRange(t_2, T, \mathcal{A})$
6: **return** $\{s \mid s = s_1 + s_2, s_1 \in \alpha_1, s_2 \in \alpha_2\}$
7: **else if** $y = t_1 - t_2$ **then**
8: $\alpha_1 = ComputeSuperRange(t_1, T, \mathcal{A}), \alpha_2 = ComputeSuperRange(t_2, T, \mathcal{A})$
9: **return** $\{s \mid s = s_1 - s_2, s_1 \in \alpha_1, s_2 \in \alpha_2\}$
10: **else if** $y = t_1 \times t_2$ **then**
11: $\alpha_1 = ComputeSuperRange(t_1, T, \mathcal{A}), \alpha_2 = ComputeSuperRange(t_2, T, \mathcal{A})$
12: **return** $\{s \mid s = s_1 \times s_2, s_1 \in \alpha_1, s_2 \in \alpha_2\}$
13: **else if** $t = Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\}$ **then**
14: **return** $\{0, \cdots, |D_{\bar{x}}|\}$
15: **else if** $t = Max_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M\}$ **then**
16: $\alpha_1 = ComputeSuperRange(t_1, T, \mathcal{A})$
17: **return** $\alpha_1 \cup \{d_M\}$
18: **else if** $t = Min_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M\}$ **then**
19: $\alpha_1 = ComputeSuperRange(t_1, T, \mathcal{A})$
20: **return** $\alpha_1 \cup \{d_M\}$

The next two propositions assert the correctness and computational complexity of Algorithm 7.

Proposition 15 *Algorithm 7, given term $t(\bar{x})$, an interpretation for type T and σ-structure $\mathcal{A}$, computes and returns set of integers α, such that α is superset of all possible values $t(\bar{x})$ may take under all possible assignments mapping each variable occurring in $\bar{x}$ to an element in T, and all the structures expanding $\mathcal{A}$. The running time of the algorithm is polynomial with respect to the size of instance structure.*

Proof: Before we start the proof, we remind readers that, as stated in Remark 1, the size of encoding of a problem instance for Enfragmo, with a single type T, is $\Omega(|T|)$. Also, recall that the formula describing the problem is fixed, and so it has a constant size. Therefore all the terms in the problem specification also have a constant size.

The proof is based on the structural induction on the structure of term, $t(\bar{x})$. There are two base cases:

- Term t is a variable: all the values a variable may take under all assignments (and all interpretations) is a subset or equal to T.
- Term t is a Count aggregate: $Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\}$ counts how many formulas out of a set of formulas, with size $|D_{\bar{x}}|$, are mapped to *true* by any assignment and interpretation. So, the value of term t is certainly greater than or equal to zero, and it is less than or equal to $|D_{\bar{x}}| = |T|^{|\bar{x}|}$.

The proof for inductive step is straightforward: Here, we prove the inductive step for $t = t_1 + t_2$. The idea in the proofs for the other cases is similar.

Let α_1 (α_2) be a superset of the values term t_1 (t_2, respectively) may take over all possible assignments and structures, computed by Algorithm 7. This algorithm returns $S = \{s \mid s = s_1 + s_2, s_1 \in \alpha_1, s_2 \in \alpha_2\}$. To show the correctness of Algorithm 7, we must show that there is no assignment γ and structure $\mathcal{B}$ expanding $\mathcal{A}$, such that $t^{\mathcal{B}}[\gamma] \notin S$. Let's assume that is not the case. So, there must be assignment γ and structure $\mathcal{B}$, such that $t^{\mathcal{B}}[\gamma] = t_1^{\mathcal{B}}[\gamma] + t_2^{\mathcal{B}}[\gamma] \notin S$. We know, by induction hypothesis, that $t_1^{\mathcal{B}}[\gamma] \in \alpha_1$, $t_2^{\mathcal{B}}[\gamma] \in \alpha_2$, and so, the integer $s = t_1^{\mathcal{B}}[\gamma] + t_2^{\mathcal{B}}[\gamma]$ must be a member of set S, and hence there cannot be such γ and $\mathcal{B}$.

The proof for the running time of Algorithm 7 is similar to that of Proposition 1. ■

Proposition 16 *In the restriction of the input language of Enfragmo which does not have Sum aggregate, language $\mathcal{L}'$, SuperRanges of all terms, computed using Algorithm 7, have polynomially many members with respect to the size of representation of instance structure $\mathcal{A}$. The maximum and minimum values a term may take have a polynomial size with respect to the size of representation of the instance structure $\mathcal{A}$.*

Proof: The proof is based on structural induction on the parse tree of term, $t(\bar{x})$.

Since in a problem instance, type T is described using a unary predicate, the size of representation of the instance structure $\mathcal{A}$ is $\Omega(|T| \log |T|)$. So $|T|^k$, where k is a constant integer, is bounded by a polynomial with respect to the size of representation of the instance structure.

The base cases are when t is a variable or a Count aggregate. In both cases, the size of the SuperRange of t is polynomial with respect to the size of the instance structure. In addition, the values of the minimum and maximum elements in the SuperRange of t are polynomial with respect to the size T.

Proof for the inductive step: Here we prove the correctness only for addition operator. The proofs for the other cases are similar.

Let $t = t_1 + t_2$, $S_1 = ComputeSuperRange(t_1, T, \mathcal{A})$ and
$S_2 = ComputeSuperRange(t_2, T, \mathcal{A})$.

By inductive hypothesis, we know that $|S_i|$ is polynomial with respect to the size of $\mathcal{A}$, $i = 1, 2$. Let $S = ComputeSuperRange(t, T, \mathcal{A})$ computed using Algorithm 7. Using the construction, we have $|S| \leq |S_1||S_2|$. Since both $|S_1|$ and $|S_2|$ are bounded by polynomials with respect to the size of representation of $\mathcal{A}$, their product is also bounded by such a polynomial.

We use $Min(T)(Max(T))$, where T is a set of integers, to represent the minimum value (maximum value, respectively) in set T. By inductive hypothesis, we know that both $Min(S_i)$ and $Max(S_i))$ are bounded by polynomials with respect to the size of representation of $\mathcal{A}$, $i = 1, 2$. The minimum value in set S is $Min(S_1) + Min(S_2)$ which is bounded by a polynomial with respect to the size of representation of $\mathcal{A}$. Moreover the maximum value in S is $Max(S_1) + Max(S_2)$ which is also bounded by a polynomial with respect to the size of representation of $\mathcal{A}$.

■

Example 18 *This example demonstrates why we excluded the Sum aggregate from the other aggregates in Proposition 16. Assume term t is $Sum_x\{f(x) : E(x)\}$ and let the corresponding type to variable x be $\{1, \cdots, 100\}$, E an expansion predicate and f an instance function, such that*

$f^{\mathcal{A}} = \{1 \mapsto 2, 2 \mapsto 4, \cdots, 100 \mapsto 2^{100}\}$. *There are* 2^{100} *ways of expanding* $\mathcal{A}$ *to structure* $\mathcal{B}$*, and in each of them,* t *evaluates to a different integer, so the SuperRange of* t *has size* 2^{100}*, while structure* $\mathcal{A}$ *can be encoded using* 100^2 *bits.*

In the rest of this section, we assume that Sum aggregate is not allowed in our language. In Section 6.6, we describe how to modify the notion of an answer to a term, in order to achieve a polynomial time grounding in the presence of the Sum aggregate.

The following construction describes one approach for computing an answer to a term, satisfying properties of Definition 20.

Construction 6 (Answers to Terms) *Let* $\mathcal{R}$ *be the pair* $(\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$*, and* t *a term. Assume that* $t_1, \ldots t_m$ *are terms, and* $\mathcal{R}_1, \ldots \mathcal{R}_m$ *(respectively) are answers to these terms with respect to* $\mathcal{A}$*. Also, let* $\mathcal{S}$ *be an answer to* ϕ *with respect to* $\mathcal{A}$*. Then we construct* $\mathcal{R}$ *as an answer to term* t *with respect to* $\mathcal{A}$ *as follows:*

(1) t *is an evaluable term,i.e., its vocabulary is a subset of instance structure vocabulary, then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$ *and for all* $n \in \alpha_{\mathcal{R}}$*, set* $\delta_{\mathcal{R}}(\gamma, n) = \top$ *iff* $t^{\mathcal{A}}[\gamma] = n$ *and set* $\delta_{\mathcal{R}}(\gamma, n) = \bot$ *iff* $t^{\mathcal{A}}[\gamma] \neq n$*.*

(2) t *is a term in the form of* $t_1 + t_2$*, then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$ *and* $\beta_{\mathcal{R}}(n) = \cup_{j \in \alpha_{\mathcal{R}_1},\, k \in \alpha_{\mathcal{R}_2},\, n=j+k} \beta_{\mathcal{R}_1}(j) \bowtie \beta_{\mathcal{R}_2}(k)$*.*

(3) t *is a term in the form of* $t_1\{-, \times\}t_2$*, similar to case (2);*

(4) t *is a term in the form of* $f(t_1, \cdots, t_m)$*, where* f *is an instance function, then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$*,* $\beta_{\mathcal{R}}(n) = \cup_{a_1 \in \alpha_{\mathcal{R}_1}, \ldots, a_m \in \alpha_{\mathcal{R}_m},\, s.t. f(a_1,\ldots,a_m)=n} \beta_{\mathcal{R}_1}(a_1) \bowtie \cdots \bowtie \beta_{\mathcal{R}_m}(a_m)$*.*

(5) t *is a term in the form of* $f(t_1, \cdots, t_m)$*, where* f *is an expansion function, then we introduce an expansion predicate* $E_f(\bar{x}, y)$ *for each expansion function* $f(\bar{x})$ *where* y *has the same sort as the range of* f*. Then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$*, and*

$$\beta_{\mathcal{R}}(n) = \cup_{a_1 \in \alpha_{\mathcal{R}_1}, \ldots, a_m \in \alpha_{\mathcal{R}_m}} \beta_{\mathcal{R}_1}(a_1) \bowtie \cdots \bowtie \beta_{\mathcal{R}_m}(a_m) \bowtie \mathcal{T}_{a_1,\cdots,a_m,n},$$

where $\mathcal{T}_{a_1,\cdots,a_m,n}$ *is an answer to* $\exists \bar{x} \wedge_i x_i = a_i \wedge y = n \wedge E_f(x_1, \cdots, x_m, y)$*.*

(6) t *is* $Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\}$*, then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$*, and we set* $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ *to be* $COUNT_{\mathcal{S},n}(\gamma)$*, where* $COUNT_{\mathcal{S},n}(\bar{y})$ *is a fresh predicate, in structure* $\mathcal{B}$*. We define* $COUNT_{\mathcal{S},n}(\gamma)$*, for every* γ*, to be equivalent to:*

$$\Big(\exists \bar{w}_1 \cdots \bar{w}_n\, (\bar{w}_1 < \bar{w}_2) \wedge \cdots \wedge (\bar{w}_{n-1} < \bar{w}_n) \wedge \phi(\bar{w}_1, y) \wedge \cdots \wedge \phi(\bar{w}_n, y) \\ \wedge \forall \bar{w} \big(\bar{w} \neq \bar{w}_1 \wedge \cdots \wedge \bar{w} \neq \bar{w}_n \rightarrow \neg \phi(\bar{w}, y)\big)\Big)[\gamma], \tag{6.1}$$

where $\bar{x} < \bar{y}$ *is a shorthand for*

$$(x_1 < y_1) \vee (x_1 = y_1 \wedge x_2 < y_2) \vee \cdots (x_1 = y_1 \wedge \cdots \wedge x_{|x|-1} = y_{|y|-1} \wedge x_{|x|} < y_{|y|}).$$

(7) t *is* $Max_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M)\}$*, then* $\alpha_{\mathcal{R}} = ComputeSuperRange(t)$*, and set* $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ *to be* $MAX_{\mathcal{T}_1,\mathcal{S},n}(\gamma)$*, where* $MAX_{\mathcal{T}_1,\mathcal{S},n}(\bar{y})$ *is a fresh predicate, in structure* $\mathcal{B}$*. We define*

$MAX_{\mathcal{T}_1,\mathcal{S},n}(\gamma)$, for every γ, to be equivalent to:

$$\Big(\big(\exists \bar{x}\; t_1(\bar{x},\bar{y}) = n \wedge \phi(\bar{x},\bar{y})\big) \wedge \forall \bar{x}(\phi(\bar{x},\bar{y}) \Rightarrow t_1(\bar{x},\bar{y}) \leq n)\Big) \vee \Big(n = d_M \wedge \forall \bar{x}\; \neg\phi(\bar{x},\bar{y})\Big). \tag{6.2}$$

(8) t is Min$_{\bar{x}}\{t_1(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m)\}$, $\alpha_{\mathcal{R}}$ and $\beta_{\mathcal{R}}$ are defined similar to case (7).

Intuitively, Equation 6.1 asserts that there are exactly n different assignments to $\bar{x}$; the values assigned to $\bar{w}_1, \cdots \bar{w}_n$, for which $\phi(\bar{x},\bar{y})$ are *true* and so, it describes the necessary and sufficient conditions for $\delta_{\beta_{\mathcal{B}}(n)}(\gamma)$. Although using our grounding algorithm we can compute, given $\mathcal{S}$, n and γ, a ground formula equivalent to Equation 6.1, in practice, many of the entries in $\beta_{\mathcal{R}}(n)$ will be eliminated during grounding as they are conjuncted with *false* or disjuncted with *true*. To reduce the grounding time, we expand the vocabulary of $\mathcal{B}$ by predicate $COUNT_{\mathcal{S},n}$, for $0 \leq n \leq |D_{\bar{x}}|$ and each occurrence of the Count aggregate. In Chapter 9, we explain how atomic formula $COUNT_{\mathcal{S},n}(\gamma)$ corresponds to a cardinality constraint and describe several approaches to translate this formula to CNF.

Equation 6.2 asserts that there is at least one assignment to $\bar{x}$ for which both $\phi(\bar{x},\bar{y})$ is *true* and $t_1(\bar{x},\bar{y}) = n$, and for all other instantiations of $\bar{x}$, either $\phi(\bar{x},\bar{y})$ is *false* or $t_1(\bar{x},\bar{y})$ is less than or equal to n. Similar to $COUNT_{\mathcal{S},n}(\bar{y})$, Enfragmo postpones describing $MAX_{\mathcal{T}_1,\mathcal{S},d_M,n}(\bar{y})$ until the MakeCNF phase. In Chapter 8, we explain how the atomic formula $MAX_{\mathcal{T}_1,\mathcal{S},d_M,n}(\gamma)$ can be translated into CNF such that it is equivalent to Formula 6.2.

Proposition 17 *The Construction 6 is correct meaning the pair $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ computed by the construction satisfies the conditions described in Definition 20.*

Proof: Since we proved the correctness of Algorithm 7, we know if $n \notin \alpha_{\mathcal{R}}$, there is no γ and no $\mathcal{B}$ expanding $\mathcal{A}$ such that $t^{\mathcal{B}}[\gamma] = n$. Thus based on Definition 20, we only need to show that for any structure $\mathcal{B}$ expanding instance structure $\mathcal{A}$, we have $\mathcal{B} \models \delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ iff $t^{\mathcal{B}}[\gamma] = n$.

The proof is by structural induction on the structure of term t (We present the proof for some of the terms, the proofs for the other cases are similar).

There are three base cases:

1. t is a variable, then the correctness of this case is trivial.
2. t is in form f, where f is a constant from instance structure. Let's assume $f^{\mathcal{A}} = m$. Function f does not have any arguments, so we have:
   - $n = m$: $\delta_{\beta_{\mathcal{R}}(n)}(\gamma) = \top$,
   - $n \neq m$: $\delta_{\beta_{\mathcal{R}}(n)}(\gamma) = \bot$.
3. t is in form f, where f is a constant from expansion structure. Let's assume Enfragmo uses expansion predicate E_f to describe f. Function f does not have any arguments, so $E_f(y)$ is a unary predicate such that for all n, $E_f(n)$ is *true* iff $f[\gamma] = n$. Based on the construction, $\delta_{\mathcal{T}_n}(\gamma) \Leftrightarrow E_f[\gamma]$. So, for every γ, we have
$$\delta_{\beta_{\mathcal{R}}(n)}(\gamma) = \delta_{\mathcal{T}_n}(\gamma) \Leftrightarrow E_f[\gamma] \Leftrightarrow f[\gamma] = n.$$
4. t is in form Count$_{\bar{x}}\{\phi(\bar{x},\bar{y})\}$. Then the construction sets $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ to be $COUNT_{\mathcal{S},n}(\gamma)$ and we defined $COUNT_{\mathcal{S},n}(\gamma)$ to be an atomic formula equivalent to $(Count_{\bar{x}}\{\phi(\bar{x},\bar{y})\} = n)\,[\gamma]$.

The proof for the inductive step is as follows:

1. Let $t(\bar{x})$ be a term in form $t_1(\bar{x}) + t_2(\bar{x})$, and $\mathcal{R}_i = (\alpha_{\mathcal{R}_i}, \beta_{\mathcal{R}_i})$ be an answer to term t_i, for $i = 1$ and 2.
   If $t^{\mathcal{B}}[\gamma]$ evaluates to n, then there must be a value j such that $t_1^{\mathcal{B}}[\gamma] = j$ and $t_2^{\mathcal{B}}[\gamma] = n - j$. By induction hypothesis, we know that if $j \notin \alpha_{\mathcal{R}_1}$, ${t_1}^{\mathcal{B}}[\gamma]$ cannot evaluate to j, therefore we simply look at $j \in \alpha_{\mathcal{R}_1}$. So, the following is the necessary and sufficient condition for $t^{\mathcal{B}}[\gamma]$ to evaluate to n:
   $$\bigvee_{j \in \alpha_{\mathcal{R}_1}} \delta_{\beta_{\mathcal{R}_1}}(\gamma, j) \wedge \delta_{\beta_{\mathcal{R}_2}}(\gamma, n - j),$$
   which is the formula corresponding to $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$, if we evaluate the result of union specified in case 2.
2. Let t be a term in the form of $f(t_1, \cdots, t_m)$, where f is an instance function. Then the above construction sets $\beta_{\mathcal{R}}(n)$ such that it contains the combination of all possible ways that f can be evaluated as n.
3. Let t be a term in the form of $f(t_1, \cdots, t_m)$, where f is an expansion function. Then the above construction set $\beta_{\mathcal{R}}(n)$ such that it asserts that $f(t_1, \cdots, t_m)$ evaluates as n under assignment γ iff $t_i[\gamma] = a_i$, $1 \leq i \leq m$, and $E_f(a_1, \cdots, a_m, n)$.
4. Let t be a term in form $\text{Max}_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M)\}$. Then the construction sets $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ to be $MAX_{\mathcal{T}_1,\mathcal{S},n}(\gamma)$ and we defined $MAX_{\mathcal{T}_1,\mathcal{S},n}(\gamma)$ to be an atomic formula equivalent to $\big(Max_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_M\} = n\big)[\gamma]$.

■

As explained in Construction 6, Enfragmo introduces some new predicates during the grounding of complex terms. These predicates can be seen as place holders for their corresponding formulas. In Chapters 8 and 9, we describe several approaches for expressing these special predicates in CNF.

1. Place holder for $Count$ aggregate: We use $COUNT_{\mathcal{S},n}(\gamma)$ for the formula corresponding to $\delta_{\mathcal{R}}(\gamma, n)$, in Case 6 of Construction 6.
2. Place holder for Max aggregate: We use $MAX_{\mathcal{T}_1,\mathcal{S},d_M,n}(\gamma)$ for the formula corresponding to $\delta_{\mathcal{R}}(\gamma, n)$, in Case 7 of Construction 6.
3. Place holder for Min aggregate: We use $MIN_{\mathcal{T}_1,\mathcal{S},d_m,n}(\gamma)$ for the formula corresponding to $\delta_{\mathcal{R}}(\gamma, n)$, in Case 8 of Construction 6.

In addition to making the grounding phase faster, this design has another benefit. If we decide to use a SAT solver which is capable of handling the aggregates natively, the place holders can be translated to the appropriate constraints, in the MakeCNF phase.

Example 19 *(Continuation of Example 17) Following Construction 6, $\beta_{\mathcal{R}}(2)$ is an extended relation describing $W_l = 2$. Since $W_l^{\mathcal{A}} = 2$, this extended relation has a single row, with true attached to it. $\beta_{\mathcal{S}_1(1)}$ is an extended $\{l_1\}$-relation representing an answer to $l_1 = 1$. Extended $\{l_1\}$-relations $\beta_{\mathcal{T}_1(4)}$ and extended $\{l_2\}$-relations $\beta_{\mathcal{T}_2(2)}$ are demonstrated in the following tables. We compute $\beta_{\mathcal{R}'}(2)$ as $\beta_{\mathcal{T}_1}(4) \bowtie \beta_{\mathcal{T}_2}(2) \cup \beta_{\mathcal{T}_1}(7) \bowtie \beta_{\mathcal{T}_2}(5)$. In other words, the answer to r' is 2 if either $t_1 = 4 \wedge t_2 = 2$ or $t_1 = 7 \wedge t_2 = 5$.*

ψ
$\top$
$D = \bot$

(a) True/False table representing $\beta_{\mathcal{R}}(2)$.

l_1	ψ
1	$\top$
$D = \bot$	

(b) True/False table representing $\beta_{\mathcal{S}_1}(1)$.

l_1	ψ
3	$\top$
$D = \bot$	

(c) True/False table representing $\beta_{\mathcal{T}_1}(4)$.

l_2	ψ
1	$\top$
$D = \bot$	

(d) True/False table representing $\beta_{\mathcal{T}_2}(2)$.

l_1	ψ
2	$\top$
$D = \bot$	

(e) True/False table representing $\beta_{\mathcal{T}_1}(7)$.

l_2	ψ
$D = \bot$	

(f) True/False table representing $\beta_{\mathcal{T}_2}(5)$.

l_1	l_1	ψ
3	1	$\top$
$D = \bot$		

(g) True/False table representing $\beta_{\mathcal{R}'}(2)$.

6.5.1 Grounding Atomic Formulas in the Presence of Complex Terms

So far, we have formally described how to compute answers to different kinds of terms. Now, we need to enhance our previous grounding algorithm, such that it is able to compute answer to an atomic formula from the answers to its terms.

As explained in Subsection 6.2.1, Enfragmo rewrites the specification such that the atomic formulas in a specification are one of the following:

1. $P(x_1, \cdots, x_k)$, where P is a arity k predicate;
2. $y = t(\bar{x})$, where all variables are associated with type T, and t is a term;
3. $t_1(\bar{x})$ *op* $t_2(\bar{x})$, where all variables are associated with type T, t_1 and t_2 are terms and $op = \{<, =\}$.

All the terms in the first kind of atomic formulas are variables, and Enfragmo uses the grounding algorithm introduced in the previous chapter to compute answers to this kind of formula. The answer to an atomic formula of the form $y = t(\bar{x})$ is a Z-relation $\mathcal{T}$, where $Z = \{z \mid z \text{ occurring in } \bar{x}\ \} \cup \{y\}$. Let $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ be an answer to variable y and $\mathcal{S} = (\alpha_{\mathcal{S}}, \beta_{\mathcal{S}})$ be an answer term $t(\bar{x})$. Then Enfragmo computes $\mathcal{T}$ as follows:

$$\mathcal{T} = \bigcup_{n \in \alpha_{\mathcal{R}} \cap \alpha_{\mathcal{S}}} \beta_{\mathcal{R}}(n) \bowtie \beta_{\mathcal{S}}(n). \tag{6.3}$$

Intuitively, Equation 6.3 says y is equal to $t(\bar{x})$ iff they both evaluate to the same integer n.

The idea to compute an answer to an atomic formula of the form $t_1(\bar{x}) = t_2(\bar{x})$ or $t_1(\bar{x}) < t_2(\bar{x})$ is similar:

Let $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ be an answer to term $t_1(\bar{x})$ and $\mathcal{S} = (\alpha_{\mathcal{S}}, \beta_{\mathcal{S}})$ be an answer term $t_2(\bar{x})$. Then Enfragmo computes extended X-relation $\mathcal{T}$, $X = \{x \mid x \text{ occurring in } \bar{x}\}$, as an answer to $t_1(\bar{x}) = t_2(\bar{x})$ as follows:

$$\mathcal{T} = \bigcup_{n \in \alpha_{\mathcal{R}} \cap \alpha_{\mathcal{S}}} \beta_{\mathcal{R}}(n) \bowtie \beta_{\mathcal{S}}(n).$$

To compute an answer to an atomic formula in form $t_1(\bar{x}) < t_2(\bar{x})$, Enfragmo uses the following:

$$\mathcal{T} = \bigcup_{\substack{m \in \alpha_{\mathcal{R}}, n \in \alpha_{\mathcal{S}} \\ \text{s.t. } m < n}} \beta_{\mathcal{R}}(m) \bowtie \beta_{\mathcal{S}}(n).$$

Example 20 *(Continuation of Example 17) Although, ψ does not have any complex terms, to demonstrate the handling of base cases, the process of computing an answer for ψ is described here. We have computed answers to r' and r. To compute an answer to $\psi(l_1, l_2) = W(l_1) - W(l_2) \leq W_l$, Enfragmo computes the following union:*

$$\left(\bigcup_{\substack{n \in \{2\}, \\ m \in \{-5,-3,-2,0\}}} \beta_{\mathcal{R}'(m)} \bowtie \beta_{\mathcal{R}(n)} \right) \cup \left(\beta_{\mathcal{R}'(2)} \bowtie \beta_{\mathcal{R}(2)} \right).$$

6.6 Answers to Terms - Part 2

As Example 18 demonstrated, there are specifications in which the sizes of SuperRanges for Sum aggregates are exponential with respect to the size of the instance structure. Remembering that our goal is to construct a polynomial time grounding framework, we need to find more efficient ways for dealing with specifications with the Sum aggregates.

In order to achieve polynomial time grounding, we define a new notion of an answer to a term, which we call it *binary answer to a term*. In this section, we use *unary answer to term* to refer the notion of answer to term introduced in Section 6.5.

Proposition 18 *The length of binary representation of the values a term may take, under different assignments and possible expansions of instance structure, is polynomial with respect to the size of representation of instance structure.*

Proof: Let $t(\bar{x})$ be a valid term in the input language of Enfragmo, and $\mathcal{A}$ be the instance structure. Using Proposition 1, we know that the size of binary representation of the smallest/largest possible value t may take under all possible assignments to $\bar{x}$ in all possible expansions of $\mathcal{A}$ is polynomial with the size of the active domain. Since the size of active domain is strictly less than the size of representation of the instance structure, every integer between the minimum and maximum values, inclusive, is representable by polynomially many bits with respect to the size of the instance structure. ■

Let t be a term with free variables $\bar{x}$, and $\mathcal{A}$ a σ-structure. Let $\mathcal{R}$ be the tuple $(m_{\mathcal{R}}, M_{\mathcal{R}}, \alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ such that $\alpha_{\mathcal{R}}$ is an extended X-relation, where $X = \{x \mid x \text{ occurring in } \bar{x}\}$, and, $\beta_{\mathcal{R}}$ is a function mapping integers to extended X-relations, $m_{\mathcal{R}}$ and $M_{\mathcal{R}}$ are two integers.

Let $\mathcal{B}$ be an expansion of $\mathcal{A}$. We define $\mathcal{R}$, for term t, such that if $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ is evaluated to *true* under structure $\mathcal{B}$, $t^{\beta}[\gamma] \geq 0$. And if $\mathcal{B}$ evaluates $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ to *false*, term $t[\gamma]$ evaluates to a negative number under structure $\mathcal{B}$. Extended X-relation $\beta_{\mathcal{R}}(a)$ represents a table such that $\delta_{\beta_{\mathcal{R}}(a)}(\gamma)$ is the necessary and sufficient condition for $t^{\mathcal{B}}[\gamma]$ to evaluate to a number whose a-th bit, in the binary representation, is one. We define $\beta_{\mathcal{R}}(a) = \emptyset$ if the a-th bit of t is always 0. We may also use $\delta_R(\gamma, n)$ and $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ interchangeably. The two integers in $\mathcal{R}$, e.g., $m_{\mathcal{R}}$ and $M_{\mathcal{R}}$, represent the minimum and maximum values $t(\bar{x})$ may take under all possible assignments to $\bar{x}$ and all possible $\mathcal{B}$. We use $m_{\mathcal{R}}$ and $M_{\mathcal{R}}$ to determine how many valuable bits t has.

Definition 21 (Binary Answer to Term t wrt $\mathcal{A}$) *We say that $\mathcal{R} = (m_{\mathcal{R}}, M_{\mathcal{R}}, \alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ is an answer to term $t(\bar{x})$ with respect to $\mathcal{A}$ iff $M_{\mathcal{R}}$ ($m_{\mathcal{R}}$) is larger than or equal to the maximum value (less than or equal to the minimum value, respectively), term t may take under all possible assignments and all expansion of structure $\mathcal{A}$, extended X-relation $\alpha_R(\gamma)$, where $X = \{x \mid x$ occurred in $\bar{x}\}$, is an answer to $t \geq 0$, and extended X-relation $\beta_{\mathcal{R}}(a)$ is an answer to the formula $\exists z \geq 0 : (2^a \leq t - 2^{a+1} \times z < 2^{a+1})$ with respect to $\mathcal{A}$.*

In Example 21, we demonstrate how a binary answer to a term, for Sum aggregate can be computed.

Example 21 *(Continuation of Example 18) Let $\mathcal{R}$ be an answer to t. We know that the maximum possible value for t is $2^{101} - 2$ and the minimum possible value for t is 0. We also know that t does not have any free variables, so all the extended relations in the answer to t must have exactly one row. As t is always non-negative, the formula corresponding to the only row in $\alpha_{\mathcal{R}}$ is $true$. The following tables show representations for $\beta_{\mathcal{R}}(0), \cdots, \beta_{\mathcal{R}}(5)$:*

ψ
$D = \bot$

(a) True/False Table for $\beta_{\mathcal{R}}(0)$.

ψ
E(1)
$D = \bot$

(b) True/False Table for $\beta_{\mathcal{R}}(1)$.

ψ
E (2)
$D = \bot$

(c) True/False Table for $\beta_{\mathcal{R}}(2)$.

ψ
E (3)
$D = \bot$

(d) True/False Table for $\beta_{\mathcal{R}}(3)$.

ψ
E (4)
$D = \bot$

(e) True/False Table for $\beta_{\mathcal{R}}(4)$.

ψ
E (5)
$D = \bot$

(f) True/False Table for $\beta_{\mathcal{R}}(5)$.

Before we explain how a binary answer to a term can be computed for different terms, we need to introduce some new operators and place holders, as in the previous section (We use $\mathcal{R}_1$ and $\mathcal{R}_2$ as binary answers to t_1 and t_2).

1. $LessEq$ operator: $LessEq(\mathcal{R}_1, \mathcal{R}_2)$ operates on two binary answers to terms, and computes an answer to a formula. The result of $LessEq$ operates is an answer to the atomic formula $|t_1| \leq |t_2|$.
2. ADD operator: $ADD(\mathcal{R}_1, \mathcal{R}_2, n)$ operates on two binary answers to terms and an integer n, and computes an answer to a formula. The result of ADD operator is an answer to the atomic formula asserting the condition under which the n-th bit in the result of adding up t_1 with t_2 is one.
3. SUB operator: $SUB(\mathcal{R}_1, \mathcal{R}_2, n)$ operates on two binary answers to terms and an integer n, and computes an answer to a formula. The result of SUB operator is an answer to the atomic formula asserting the condition under which the n-th bit in the result of subtracting t_1 from t_2 is one.
4. MUL operator: $MUL(\mathcal{R}_1, \mathcal{R}_2, n)$ operates on two binary answers to terms and an integer n, and computes an answer to a formula. The result of MUL operator is an answer to the atomic formula asserting the condition under which the n-th bit in the result of subtracting t_1 from t_2 is one.
5. EQ operator: $EQ(n, \mathcal{R}_1)$ operates on an integer n and a binary answer to a term, and computes an answer to a formula. The result of EQ operator is an answer to the atomic formula $t_1 = n$.

Construction 7 (Binary Answer to Terms) *Let $\mathcal{R}$ be $(m_{\mathcal{R}}, M_{\mathcal{R}}, \alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$, and t a term. Assume that $t_1, \cdots, t_m$ are terms, and $\mathcal{R}_1, \cdots, \mathcal{R}_m$ are answers to those terms with respect to $\mathcal{A}$, respectively. Also, let $\mathcal{S}$ be an answer to ϕ. Then $\mathcal{R}$ is a binary answer to term t with respect to $\mathcal{A}$:*

1. *If t is a variable x, then*
   (a) *Set $m_{\mathcal{R}}$ ($M_{\mathcal{R}}$) to be the minimum (the maximum, respectively) value in T.*
   (b) *Set $\delta_{\alpha_{\mathcal{R}}}(\gamma) = \top$ iff $(x \geq 0)[\gamma]$.*
   (c) *For n s.t. $\min(|m_{\mathcal{R}}|, |M_{\mathcal{R}}|) \leq 2^n \leq \max(|m_{\mathcal{R}}|, |M_{\mathcal{R}}|)$ set $\delta_{\beta_{\mathcal{R}}(n)}(\gamma) = \top$ iff the n-th bit of the binary representation of $\gamma|_x$ is one.*
   (d) *For n s.t. $\min(|m_{\mathcal{R}}|, |M_{\mathcal{R}}|) \leq 2^n \leq \max(|m_{\mathcal{R}}|, |M_{\mathcal{R}}|)$ set $\delta_{\beta_{\mathcal{R}}(n)}(\gamma) = \bot$ iff the n-th bit of the binary representation of $\gamma|_x$ is zero.*

2. *t is a term in form of $t_1 + t_2$:*
   (a) *Set $m_{\mathcal{R}}$ ($M_{\mathcal{R}}$) to be $m_{\mathcal{R}_1} + m_{\mathcal{R}_2}$ ($M_{\mathcal{R}_1} + M_{\mathcal{R}_2}$, respectively).*
   (b) *To define $\delta_{\alpha_{\mathcal{R}}}$, we use $LessEq$ operator. We set $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ to be*

$$\begin{aligned} &\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma) \wedge \delta_{\alpha_{\mathcal{R}_2}}(\gamma)\right) \\ \vee &\left(\neg\delta_{\alpha_{\mathcal{R}_1}}(\gamma) \wedge \delta_{\alpha_{\mathcal{R}_2}}(\gamma) \wedge \delta_{LessEq(\mathcal{R}_1,\mathcal{R}_2)}(\gamma)\right) \\ \vee &\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma) \wedge \neg\delta_{\alpha_{\mathcal{R}_2}}(\gamma) \wedge \delta_{LessEq(\mathcal{R}_2,\mathcal{R}_1)}(\gamma)\right). \end{aligned}$$

   (c) *To define $\delta_{\beta_{\mathcal{R}}}$, we use ADD and SUB operators. We set $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ to be*

$$\begin{aligned} &\left(\neg\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma)\ xor\ \delta_{\alpha_{\mathcal{R}_2}}(\gamma)\right) \wedge \delta_{ADD(\mathcal{R}_1,\mathcal{R}_2,n)}(\gamma)\right) \\ \bigvee &\left(\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma)\ xor\ \delta_{\alpha_{\mathcal{R}_2}}(\gamma)\right) \wedge \delta_{SUB(\mathcal{R}_1,\mathcal{R}_2,n)}(\gamma)\right). \end{aligned}$$

3. *t is a term in form of $t_1\{-, \times\}t_2$: similar to case (2);*

4. *t is a term in form of $f(x_1, \cdots, x_m)$, where f is an instance function:*
   (a) *Set $m_{\mathcal{R}}$ ($M_{\mathcal{R}}$) to be the minimum (the maximum, respectively) value in T.*
   (b) *Set $\delta_{\alpha_{\mathcal{R}}}(\gamma) = \top$ iff $f^{\mathcal{A}}[\gamma] \geq 0$.*
   (c) *Set $\delta_{\alpha_{\mathcal{R}}}(\gamma) = \bot$ iff $f^{\mathcal{A}}[\gamma] < 0$.*
   (d) *To define $\delta_{\beta_{\mathcal{R}}}$, we use the EQ operator. We set $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ to be*

$$\bigvee_{\substack{k \in \text{Range of } f^{\mathcal{A}} \text{ s.t.} \\ \text{the } n\text{-th bit of } k \text{ is one and} \\ f^{\mathcal{A}}(a_1,\cdots,a_m)=k}} \bigwedge_{i=1}^{m} \delta_{EQ(a_i,\mathcal{R}_i)}(\gamma).$$

5. *t is a term in form of $f(x_1, \cdots, x_m)$, where f is an expansion function: We introduce an expansion predicate $E_f(\bar{x}, y)$, for each expansion function $f(\bar{x})$, where y has the same sort as the range of f.*
   (a) *Set $m_{\mathcal{R}}$ ($M_{\mathcal{R}}$) to be the minimum (the maximum, respectively) value in T.*

(b) *Set* $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ *to be*

$$\bigvee_{\substack{o:o\in T \\ \text{and } o\geq 0}} E_f(\gamma|_{\bar{x}}, o).$$

(c) *To define* $\delta_{\beta_{\mathcal{R}}}$*, we use* EQ *operator. We set* $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ *to be*

$$\bigvee_{\substack{k\in T \text{ s.t.} \\ \text{the } n\text{-th bit of } k \text{ is one}}} \left(\bigwedge_{i=1}^{m} \delta_{EQ(a_i,\mathcal{R}_i)}(\gamma) \right) \wedge E_f(\gamma, n).$$

6. t *is* $Count_{\bar{x}}\{\phi(\bar{x},\bar{y})\}$*:*

(a) *Set* $m_{\mathcal{R}}$ *(*$M_{\mathcal{R}}$*) to be 0 (*$|D_{\bar{x}}|$*, respectively).*

(b) *Set* $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ *to be* $\top$*.*

(c) *To define* $\delta_{\beta_{\mathcal{R}}(n)}$ *we use* $COUNT(\mathcal{S}, o, \gamma)$ *place holder, defined in Section 6.5. We set* $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ *to be:*

$$\bigvee_{\substack{o\leq |D_{\bar{x}}| \text{ s.t.} \\ \text{the } n\text{-th bit of } o \text{ is one}}} COUNT_{\mathcal{S},o}(\gamma).$$

7. t *is* $Max_{\bar{x}}\{t_1(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_M)\}$*:*

(a) *Set* $m_{\mathcal{R}}$ *(*$M_{\mathcal{R}}$*) to be* $m_{\mathcal{R}_1}$ *(*$M_{\mathcal{R}_1}$*, respectively).*

(b) *Set* $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ *to be:*

$$\bigvee_{\gamma' \text{ is assignment to } \bar{x}} \delta_{\mathcal{S}}(\gamma', \gamma) \wedge \delta_{\alpha_{\mathcal{R}_1}}(\gamma', \gamma)$$
$$\vee \left((d_M \geq 0) \wedge \bigwedge_{\gamma' \text{ is assignment to } \bar{x}} \neg\delta_{\mathcal{S}}(\gamma', \gamma) \right).$$

(c) *Set* $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ *to be equivalent to the following expression:*

- *Either there exists assignment* γ_1*, mapping variables in* $\bar{x}$ *to elements in* T*, such that the* n*-th bit of the result of* $t_1[\gamma_1, \gamma]$ *is one and* $\phi[\gamma_1, \gamma]$ *is true, and also for all other* γ_2*, mapping variables in* $\bar{x}$ *to elements in* T*, where* $\phi[\gamma_1, \gamma]$ *is true,* $t_2[\gamma_2, \gamma]$ *is less than or equal to* $t_1[\gamma_1, \gamma]$*.*
- *Or for all* γ_1*, mapping variables in* $\bar{x}$ *to elements in* T*,* $\phi[\gamma_1, \gamma]$ *is falseand the* n*-th bit of* d_M*, in the binary representation is one.*

This condition can be expressed explicitly using the answer to ϕ*, binary answer to* t_1 *and* $LessEq$ *operator. Since it would be too long, we did not include it here.*

8. t *is* $Min_{\bar{x}}\{t_1(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m)\}$*: similar to case (7).*

9. t *is* $Sum_{\bar{x}}\{t_1(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$*:*

(a) *Set* $m_{\mathcal{R}}$ *(*$M_{\mathcal{R}}$*) to be* $\min(|A|^{|\bar{x}|} * m_{\mathcal{R}_1}, 0)$ *(*$\max(|A|^{|\bar{x}|} * M_{\mathcal{R}_1}, 0)$*, respectively).*

(b) Set $\delta_{\alpha_{\mathcal{R}}}(\gamma)$ to be equivalent to the following linear equation:

$$\sum_{\substack{\gamma' \text{ is assignment to } \bar{x} \\ 0 \leq n \leq \log(|M_{\mathcal{R}}|)}} 2^n * \chi\left(\delta_{\mathcal{T}}(\gamma, \gamma') \wedge \delta_{\beta_{\mathcal{R}_1}(n)}(\gamma, \gamma')\right) * (2 * \chi\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma, \gamma')\right) - 1) \geq 0,$$

where $\chi(\psi)$ is the characteristic function returning 1 iff ψ is evaluated to true.

(c) Set $\delta_{\beta_{\mathcal{R}}(n)}(\gamma)$ to be equivalent to the following expression:
The n-th bit in the result of the following summation is 1:

$$\sum_{\substack{\gamma' \text{ is assignment to } \bar{x} \\ 0 \leq m \leq \log(|M_{\mathcal{R}}|)}} 2^m * \chi\left(\delta_{\mathcal{T}}(\gamma', \gamma) \wedge \delta_{\beta_{\mathcal{R}_1}(m)}(\gamma', \gamma)\right) * (2 * \chi\left(\delta_{\alpha_{\mathcal{R}_1}}(\gamma', \gamma)\right) - 1),$$

where $\chi(\psi)$ is the characteristic function returning 1 iff ψ is evaluated to true.

Similar to Proposition 17, it can be shown that Construction 7 computes a correct binary answer to terms.

Analogous to Section 6.5, to reduce the grounding time, we introduce place holders when we are computing binary answers to terms:

1. Place holder $BLESSEQ_{\mathcal{R}_1,\mathcal{R}_2}(\gamma)$: We use $BLESSEQ_{\mathcal{R}_1,\mathcal{R}_2}(\gamma)$ to express the atomic formula corresponding to $\delta_{LessEq(\mathcal{R}_1,\mathcal{R}_2)}(\gamma)$.
2. Place holder $BEQ_{n,\mathcal{R}}(\gamma)$: We use $BEQ_{n,\mathcal{R}}(\gamma)$ to express the atomic formula corresponding to $\delta_{EQ(n,\mathcal{R})}(\gamma)$.
3. Place holder $BADD_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$ ($BSUB_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$, $BMUL_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$): We use $BADD_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$ ($BSUB_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$, $BMUL_{\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$) to express the atomic formula corresponding to $\delta_{ADD(\mathcal{R}_1,\mathcal{R}_2,n)}(\gamma)$ ($\delta_{SUB(\mathcal{R}_1,\mathcal{R}_2,n)}(\gamma)$, $\delta_{MUL(\mathcal{R}_1,\mathcal{R}_2,n}(\gamma)$).
4. Place holder $COUNT_{\mathcal{S},n}(\gamma)$ is the same as described in Section 6.5.
5. Place holder $BMAX_{\mathcal{S},\mathcal{R}_1,d_M,n}(\gamma)$: We use atomic formula $BMAX_{\mathcal{S},\mathcal{R}_1,d_M,n}(\gamma)$ to express that the n-th bit of the return value of Max aggregate is one.
6. Place holder $BMIN_{\mathcal{S},\mathcal{R}_1,d_m,n}(\gamma)$: We use atomic formula $BMIN_{\mathcal{S},\mathcal{R}_1,d_m,n}(\gamma)$ to express that the n-th bit of the return value of Min aggregate is one.
7. Place holder $BSUM_{\mathcal{S},\mathcal{R}_1,n}(\gamma)$: We use atomic formula $BSUM_{\mathcal{S},\mathcal{R}_1,n}(\gamma)$ to express that the n-th bit of the return value of Sum aggregate is one.

6.6.1 Grounding an Atomic Formula Using Binary Term Tables

Similar to Subsection 6.5.1, here we describe how to construct an answer to an atomic formula from binary answers to its terms.

Let t_1 and t_2 be terms and $\mathcal{R}_1$ and $\mathcal{R}_2$ be binary answers to them with respect to structure $\mathcal{A}$. Also, let $\mathcal{R}_3$ be a binary answer to term y, where y is a variable.

1. $\mathcal{R}$ is an answer to $y = t_1(\bar{x})$ with respect to $\mathcal{A}$ if

$$\mathcal{R} = LessEq(\mathcal{R}_1, \mathcal{R}_3) \bowtie LessEq(\mathcal{R}_3, \mathcal{R}_1).$$

2. $\mathcal{R}$ is an answer to $t_1(\bar{x}) = t_2(\bar{y})$ with respect to $\mathcal{A}$ if

$$\mathcal{R} = LessEq(\mathcal{R}_1, \mathcal{R}_2) \bowtie LessEq(\mathcal{R}_2, \mathcal{R}_1).$$

3. $\mathcal{R}$ is an answer to $t_1(\bar{x}) < t_2(\bar{y})$ with respect to $\mathcal{A}$ if

$$\mathcal{R} = LessEq(\mathcal{R}_1, \mathcal{R}_2).$$

6.7 Experimental Evaluation

In Chapter 3, we showed that having complex terms in system's language enables users to express complex axioms compactly. Table 3.5 in Section 3.4, shows that Enfragmo handles complex terms efficiently.

As explained in Chapter 4, we extended the input language of Enfragmo with complex terms such that it still captures NP. Our main motivation for adding support for complex terms in Enfragmo was to help users write specifications for their problems. To show how using complex terms makes the task of developing specifications easier, we express the three specifications described in Chapter 3 for the Social Golfers problem without using complex terms. We also compare the performance of Enfragmo and IDP on the three specifications we described for this problem; SG-01, SG-02 and SG-03.

Recall that all three specifications for Social golfers contain the following three axioms:

1. Each group must have GroupSize golfers in it: $\forall w, g : Count_p\{M(w, g, p)\} = GroupSize$,
2. No two golfers plays in the same group more than once: $\forall p_1 p_2 : (p_1 < p2) \rightarrow Count_{w,g}\{M(w, g, p_1) \wedge M(w, g, p_2)\} \leq 1$,
3. Each week, all golfers must play in exactly one group: $\forall w, p : Count_g\{M(w, g, p)\} = 1$.

In Chapter 3, we described three more axioms to break the symmetries in the search space:

A-1 In each week, golfer number 1 is going to play in group number one: $\forall w : M(w, 1, 1)$;

A-2 In each week, groups are ordered based on the least index of golfers who is playing in the groups: $\forall w, g_1, g_2 : Succ(g_1, g2) \rightarrow Min_p\{p; M(w, g_1, p); PlayerCount) < Min_p\{p; M(w, g_2, p); 0\}$;

A-3 The second smallest index of the golfer in the first group of week i must be less than or equal to the second smallest index of golfer in the first group of week j, for all $j > i$: $\forall w_1, w_2 : Succ(w_1, w_2) \rightarrow Min_p\{p; M(w_1, 1, p) \wedge p \neq 1; PlayerCount) < Min_p\{p; M(w_2, 1, p) \wedge p \neq 1; 0\}$.

Specification SG-01 contains only axioms (1)-(3). Specification SG-02, in addition to (1)-(3), contains A-1 and A-2. Specification SG-03 is the same as specification SG-01 plus axiom A-1, A-2 and A-3.

To express axiom 1, we can use arity 4 predicate $CountPlayer(Weeks, Group, GroupSize, Player)$, with a set of axioms asserting that for every Week w and Group g, there are exactly $GroupSize$ distinct players, $p_1, \cdots, p_{GroupSize}$, such that $M(w, g, p_i)$ is true, for $i = 1, \cdots, GroupSize$.

Axiom 2 can be expressed in pure FO logic as follows:

$$\forall p_1, p_2 : (p1 < p_2) \rightarrow \forall w_1, g_1, w_2, g_2 : (M(w_1, g_1, p_1) \wedge M(w_1, g_1, p_2) \wedge (g_1 \neq g_2) \wedge (w_1 \neq w_2)) \rightarrow \neg \left(M(w_2, g_2, p_1) \wedge M(w_2, g_2, p_2)\right).$$

The conjunction of the following two FO axioms express Axiom 3:

$$\forall w, p : \exists g : M(w, g, p).$$
$$\forall w, p, g_1, g_2 : \quad (g_1 \neq g_2) \wedge M(w, g_1, p) \rightarrow \quad \neg M(w, g_2, p).$$

Axiom A-2 can be expressed without using complex terms by the following formulas:

$$\forall w, g_1, g_2 : Succ(g_1, g_2) \rightarrow$$
$$\forall p_1, p_2 : M(w, g_1, p_1) \wedge (\forall p : M(w, g_1, p) \rightarrow (p_1 \leq p))$$
$$\wedge M(w, g_2, p_2) \wedge (\forall p : M(w, g_2, p) \rightarrow (p_2 \leq p))$$
$$\rightarrow p_1 < p_2.$$

The idea to express Axiom A-3 in the first-order logic is very similar to what has been used to express Axiom A-2.

Not only is it more difficult to develop certain specifications without using complex terms, but the performance is also an issue. We developed the first-order specifications corresponding to SG-01,SG-02 and SG-03, and tried them for solving Social Golfer instances, using both Enfragmo and IDP. As one may expect, both solvers failed to solve medium and large size instances, using first-order specifications.

Since the input languages of IDP and Enfragmo are very similar, one can translate specifications for one of the systems to specifications to the other one. In Table 6.4, we compare the performance of IDP and Enfragmo on SG-01, SG-02 and SG-03. Table 6.4 shows that Enfragmo performs better than IDP on specifications with complex terms.

To summarize the experiments, we showed that if Enfragmo did not support complex terms, developing some specifications required much effort and knowledge. Although we discussed that Enfragmo performs poorly on the first-order axioms corresponding to complex terms, we tested the first-order axioms on the IDP system, and we received the same poor performance. So, we can conclude that supporting complex terms is beneficial in two ways:

- It enables users to develop their specification more easily. Also, users can describe complex properties of their problems more conveniently in the presence of complex terms.
- It provides the user with faster solving time. To enable Enfragmo to support complex term, we have to modify/revise its grounding algorithms. The modified algorithms are efficient and do not cause a lot of computational overheads. The same observation is true for the IDP system.

Specification	Enfragmo	IDP
SG-01	126/53.2	98/46.6
SG-02	149/24.8	142/2.79
SG-03	161/19.8	142/0.17

Table 6.4: Performance of Enfragmo and IDP on the Social Golfers problem. The entry n/t means that n instances were solved out of 175 instances, with an average running time of t seconds. The 1800-second timeouts are included in the times.

6.8 Conclusion

In this chapter, we described how we extended our engine to handle complex terms. We proposed three extensions of the grounding algorithm, introduced in Chapter 5. The first extension, described in Section 6.4, is suitable for handling specifications whose terms' vocabulary is a subset of the instance structure vocabulary. We introduced *unary answer to term* in Section 6.5, and modified our grounding algorithm such that it can ground specifications with no occurrence of the Sum aggregate, in polynomial time. Finally, we introduced *binary answer to terms* and modified our grounding approach to handle all specifications expressible in the input language of Enfragmoin polynomial time.

The results of the experiments presented in Sections 6.7 and 3.4, confirm that Enfragmo handles the complex terms efficiently.

Chapter 7

Related Work

In this chapter, we review some of well-known grounding-based solvers. To investigate how a grounder works, one needs to define the syntax of its input language and the target language in which the output is generated. The input languages of grounders we study in this section, allow users to describe all problems in the NP complexity class. So almost every problem which is solvable by one of them can also be solved by the other solvers, too. What makes the comparison of these grounders difficult is the fact that the performance of a grounder (a solver) depends on how the problem has been specified in its input language. When fixing a solver and a problem, different specifications for the problem can result in different performances.

The grounders can be classified into the following three groups:

1. Bottom-Up Grounders: The answer[1] to each part of specification is computed based on the answers to its sub-parts.

2. Top-Down Grounders: These grounders start computing the answers from the axioms. The answer for each subformula is computed from the answer computed from its parent, in the parse tree of the formula.

3. Incremental Grounders (Solvers): The grounder translates a part of the input specification and solves that part of the problem. If the obtained solution satisfies the whole problem specification, the process is complete. Otherwise, the previous partial grounding will be extended and passed to the solver again. Extending the partial grounding continues until a consistent solution is found or the solver reports unsatisfiability.

7.1 Bottom-Up Grounders

7.1.1 MXG

In [54], a framework based on classical first-order logic (FO) extended with inductive definition was proposed which captures exactly the problems in NP. This framework is based explicitly on the

[1] We did not define what an answer is as it depends on the algorithm used by the grounder.

model expansion task. Patterson et al., in [58], proposed a grounding approach for the proposed framework.

The grounding task is to produce a ground formula $\psi = Gnd(\phi,\mathrm{A})$, such that models of ψ correspond to solutions for instance $\mathcal{A}$. Formally, to ground, the grounder brings domain elements into the syntax by expanding the vocabulary with a new constant symbol for each element of the domain. For domain A, we denote the set of such constants by $\tilde{A}$.

Example 22 *(Continuation of Example 1) Formula ϕ in the first-order logic together with fixed instance vocabulary σ, constitutes a specification for the graph 3-Colouring problem where the colour of some of the vertices are fixed:*

$$\begin{aligned}
&\forall x \Big[\big(R_p(x) \rightarrow R(x)\big) \wedge \big(B_p(x) \rightarrow B(x)\big) \wedge \big(G_p(x) \rightarrow G(x)\big)\Big] \\
&\wedge \forall x \Big[\big(R(x) \vee B(x) \vee G(x)\big)\Big] \\
&\wedge \forall x \Big[\big(R(x) \rightarrow \neg B(x)\big) \wedge \big(R(x) \rightarrow \neg G(x)\big) \wedge \big(B(x) \rightarrow \neg G(x)\big)\Big] \\
&\wedge \forall x \forall y \Big[\big(E(x,y) \vee E(y,x)\big) \rightarrow \\
&\qquad \Big(\neg\big(R(x) \wedge R(y)\big) \wedge \neg\big(B(x) \wedge B(y)\big) \wedge \neg\big(G(x) \wedge G(y)\big)\Big)\Big].
\end{aligned}$$

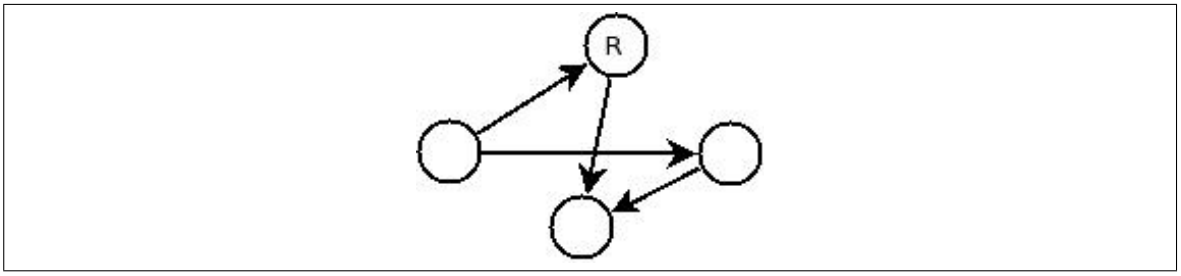

Figure 7.1: Graph Colouring Instance

Assuming the input graph is the one represented in Fig 7.1, the instance structure for vocabulary $\sigma = \{E, R_p, G_p, B_p\}$, is $E^{\mathcal{A}} = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)\}$, $R_p^{\mathcal{A}} = \{v_1\}$ and $G_p^{\mathcal{A}} = B_p^{\mathcal{A}} = \{\}$. The task is to find an expansion $\mathcal{B}$ of $\mathcal{A}$ that satisfies ϕ:

$$(\overbrace{V; E^{\mathcal{A}}, {R_p}^{\mathcal{A}}, {G_p}^{\mathcal{A}}, {B_p}^{\mathcal{A}}}^{\text{Instance}}, \overbrace{R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}}}^{\text{Solution}})_{\mathcal{B}} \models \phi.$$

Every interpretation of the expansion vocabulary $\varepsilon = \{R, B, G\}$, for structures $\mathcal{B}$ that satisfies ϕ, is a proper 3-colouring for G which extends the given partial colouring.

Definition 22 (Reduced Grounding for MX) *([58]) Formula ψ is a* reduced grounding *of formula ϕ over σ-structure $\mathcal{A} = (A, \sigma^{\mathcal{A}})$ if (1) ψ is a ground formula over $\varepsilon \cup \tilde{A}$; and (2) for every expansion structure $\mathcal{B} = (A, \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$ over $vocab(\phi)$, $\mathcal{B} \models \phi$ iff $(\mathcal{B}, \tilde{A}^{\mathcal{B}}) \models \psi$, where $\tilde{A}^{\mathcal{B}}$ denotes the interpretation of the new constants $\tilde{A}$.*

During grounding, the symbols of the instance vocabulary are "evaluated out" and a reduced grounding is obtained.

Proposition 19 *([58]) Let ψ be a reduced grounding of ϕ over σ-structure $\mathcal{A}$. Then $\mathcal{A}$ can be expanded to a model of ϕ iff ψ is satisfiable.*

Thus, through grounding, the model expansion problem is reduced (in polynomial time) to propositional satisfiability.

The following algorithm, which is a bottom-up process that parallels database query evaluation, produces a grounding for a given specification. The process constructs, for each sub-formula $\phi(\bar{x})$ with free variables $\bar{x}$, the set of reduced groundings for ϕ under all possible instantiations of $\bar{x}$. A representation of this set is called an *answer to $\phi(\bar{x})$ with respect to $\mathcal{A}$*.

An extended X-relation is a relation associated with tuple of variables X. This grounding produces/maintains extended relations such that each tuple $\gamma : X \mapsto A$ occurs in exactly one pair (γ, ψ), where ψ is a formula.

Definition 23 (extended X-relation; function $\delta_{\mathcal{R}}$) *([58]) 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 $\gamma : X \to A$; (2) for every γ, there is at most one ψ s.t. $(\gamma, \psi) \in \mathcal{R}$. The* function represented by *$\mathcal{R}$, is a mapping from tuples γ of elements of the domain $\mathcal{A}$ to formulas, defined as:*

$$\delta_{\mathcal{R}}(\gamma) = \begin{cases} \psi & \text{if } (\gamma, \psi) \in \mathcal{R}, \\ \bot & \text{if there is no pair } (\gamma, \psi) \in \mathcal{R}. \end{cases}$$

Definition 24 (Answer to ϕ with respect to $\mathcal{A}$) *([58]) Let ϕ be a formula in $\sigma \cup \varepsilon$ with free variables X, $\mathcal{A}$ be a σ-structure with domain A, and $\mathcal{R}$ an extended X-relation over $\mathcal{A}$. If $\mathcal{R}$ is* an answer to ϕ with respect to $\mathcal{A}$ *for any $\gamma : X \to A$, we have that $\delta_{\mathcal{R}}(\gamma)$ is a reduced grounding of $\phi[\gamma]$ over $\mathcal{A}$. Here, $\phi[\gamma]$ denotes the result of instantiating free variables in ϕ according to γ.*

Example 23 *([58]) Let $\phi = \exists x \exists y \exists z \phi'$ where $\phi' = P(x, y, z) \wedge E(x, y) \wedge E(y, z)$, $\sigma = \{P\}$, and $\varepsilon = \{E\}$. Let $\mathcal{A}$ be a σ-structure such that $P^{\mathcal{A}} = \{(1, 2, 3), (3, 4, 5)\}$. Then this extended relation $\mathcal{R}$*

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

is an answer to ϕ' with respect to $\mathcal{A}$. It is easy to see, for example, that $\delta_{\mathcal{R}}(1, 2, 3) = E(1, 2) \wedge E(2, 3)$ is a reduced grounding of $\phi'[(1, 2, 3)] = P(1, 2, 3) \wedge E(1, 2) \wedge E(2, 3)$, and $\delta_{\mathcal{R}}(1, 1, 1) = \bot$ is a reduced grounding of $\phi'[(1, 1, 1)]$. The following extended relation is an answer to $\phi'' = \exists z \phi'$.

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 $\phi''[(1,2)]$. Finally, the following represents an answer to ϕ, where the single formula is a reduced grounding of ϕ.

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

The standard relational algebra has the following operations, each corresponding to a connective in FO: complement (negation); join (conjunction); union (disjunction), project (existential quantification); and divide (universal quantification). Extended X-relations can be generalized as follows:

Definition 25 (Extended Relational Algebra Operations) *([58]) Let $\mathcal{R}$ be an extended X-relation and $\mathcal{S}$ an extended Y-relation, both over domain A.*

1. *$\neg\mathcal{R}$ is the extended X-relation $\neg\mathcal{R} = \{(\gamma, \psi) \mid \gamma : X \to A, \delta_{\mathcal{R}}(\gamma) \neq \top,$ and $\psi = \neg\delta_r(\gamma)\}$;*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is the extended $X \cup Y$-relation $\{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in \mathcal{R}, \gamma|_Y \in \mathcal{S},$ and $\psi = \delta_{\mathcal{R}}(\gamma|_X) \wedge \delta_{\mathcal{S}}(\gamma|_Y)\}$;*
3. *$\mathcal{R} \cup \mathcal{S}$ is the extended $X \cup Y$-relation $\mathcal{R} \cup \mathcal{S} = \{(\gamma, \psi) \mid \gamma|_X \in \mathcal{R}$ or $\gamma|_Y \in \mathcal{S},$ and $\psi = \delta_{\mathcal{R}}(\gamma|_X) \vee \delta_{\mathcal{S}}(\gamma|_Y)\}$;*
4. *The Y-*projection *of $\mathcal{R}$, denoted by $\pi_Y(\mathcal{R})$, is the extended Y-relation $\{(\gamma', \psi) \mid \gamma' = \gamma|_Y$ for some $\gamma \in \mathcal{R}$ and $\psi = \bigvee_{\{\gamma \in \mathcal{R} \mid \gamma' = \gamma|_Y\}} \delta_{\mathcal{R}}(\gamma)\}$;*
5. *The Y-*quotient *of $\mathcal{R}$, denoted by $d_Y(\mathcal{R})$, is the extended Y-relation $\{(\gamma', \psi) \mid$ for all γ such that $\gamma' = \gamma|_Y \to \gamma \in \mathcal{R}$ and $\psi = \bigvee_{\{\gamma \in \mathcal{R} \mid \gamma' = \gamma|_Y\}} \delta_{\mathcal{R}}(\gamma)\}$;*

Proposition 20 *([58]) Suppose that $\mathcal{R}$ is an answer to ϕ_1 and $\mathcal{S}$ is an answer to ϕ_2, both with respect of structure $\mathcal{A}$. Then*

1. *$\neg\mathcal{R}$ is an answer to $\neg\phi_1$ wrt $\mathcal{A}$;*
2. *$\mathcal{R} \bowtie \mathcal{S}$ is an answer to $\phi_1 \wedge \phi_2$ wrt $\mathcal{A}$;*
3. *$\mathcal{R} \cup \mathcal{S}$ is an answer to $\phi_1 \vee \phi_2$ wrt $\mathcal{A}$;*
4. *If Y is the set of free variables of $\exists\bar{z}\phi_1$. Then $\pi_Y(\mathcal{R})$ is an answer to $\exists\bar{z}\phi_1$ wrt $\mathcal{A}$;*
5. *If Y is the set of free variables of $\forall\bar{z}\phi_1$. Then $d_Y(\mathcal{R})$ is an answer to $\forall\bar{z}\phi_1$ wrt $\mathcal{A}$.*

The proof is straightforward, and can be found in [58, 56].

To ground with this algebra, the answer to atomic formula $P(\bar{x})$ can be defined as follows. If P is an instance predicate, the set of tuples $\{\langle\bar{a}, \top\rangle\colon \bar{a} \in P^{\mathcal{A}}\}$. If P is an expansion predicate, the set of all tuples $\langle\bar{a}, P(\bar{a})\rangle$, for $\bar{a}$ a tuple of elements from the domain of $\mathcal{A}$. Then applying the algebra inductively on the structure of the formula results in a grounding.

Formally, extended relations representing expansion predicates are universal, but in practice one may represent them implicitly and not enumerate all the tuples. As operations are applied, some subsets of columns remain universal, while other do not. The concept of extended X-relations can be generalized to represent this idea explicitly, and call the variables which are implicitly universal "hidden".

Definition 26 (Extended Hidden X-Relation) *([56]) An extended hidden X-relation $\mathcal{R}_Y$ is an extended relation with explicit attributes X and implicit attributes Y, such that:*

Values of hidden attributes do not appear explicitly in tuples. An extended hidden relation $\mathcal{R}_Y = \{(\gamma, \psi) \mid \gamma : X \to A, \psi = \delta_{\mathcal{R}}(\gamma)\}$ *is a compact representation of extended table* $\mathcal{R}_{X \cup Y} = \{(\gamma, \psi) \mid \gamma : X \cup Y \to A, \psi = \delta_{\mathcal{R}}(\gamma|_{X \cup Y})\}$.

All the operations of the algebra generalize easily to hidden versions. The hidden variables technique does not alter the semantics of the operations.

MXG engine enters the CNF generation phase in which a CNF formula, a SAT instance, is created from the produced ground formula.

Cons and Pros

This grounding approach is based on algebraic database theory and almost all the optimization techniques used in the database community can be used in this approach, too. The other advantage of this approach is that adding a new operator (logical operator) to the syntax is easy. To do so, one simply needs to describe the effect of that operator on its subformulas as a relational algebraic formula (series of operations) on the subformulas' extended relations.

MXG works well under the assumption that the instance predicates are usually sparse and have a relatively small size. In the grounding approach used by MXG, when the algorithm encounters a negation, it needs to enumerate all the tuples which are not in that relation. As we assumed the instance predicates are relatively small, the size of interpretation of the negation of an instance predicate would be large. So, one cannot use this approach for medium and large sized domains. This grounding approach was developed for function-free FO formulas. We know this fragment of logic allows us to express all NP problems [36], but expressing certain problems is not straightforward [2].

7.1.2 KodKod

The relational logic of Alloy [46] is a mixture of first-order logic quantifiers and relational algebra operators. This logic is designed for modelling software abstractions, their properties and invariants [47].

Kodkod [66] is a grounder which translates its input problem, described in an extension of Alloy, into a SAT instance (directly) and, then applies an off-the-shelf SAT solver to the resulting propositional formula. The input language of Kodkod (see Appendix A) is an extension of Alloy in two ways: The universe of atoms for a specification is made explicit, and the value of each relation (which is sometimes called variable) is explicitly bounded from above and below, by relational constants (i.e., set of tuples). All bounding constants consist of tuples that are drawn from the same finite universe of elements. The upper bound specifies the tuples that a relation may contain; the lower bound specifies the tuples that it should contain.

Example 24 *(Continuation of Example 1) The graph colouring problem, introduced in Example 1, can be described for Kodkod by following specification:*

1. $\{v_1, v_2, v_3, v_4\}$.
2. $E :_2 [\{\langle 1,3\rangle, \langle 2,1\rangle, \langle 2,4\rangle, \langle 4,3\rangle\} : \{\langle 1,3\rangle, \langle 2,1\rangle, \langle 2,4\rangle, \langle 4,3\rangle\}]$.

[2] These two issues have been addressed and two solutions have been proposed in my works [5] and [7].

3. $V :_1 [\{\langle v_1\rangle, \langle v_2\rangle, \langle v_3\rangle, \langle v_4\rangle\} : \{\langle v_1\rangle, \langle v_2\rangle, \langle v_3\rangle, \langle v_4\rangle\}]$.
4. $R :_1 [\{\langle v_1\rangle\} : \{\langle v_1\rangle, \langle v_2\rangle, \langle v_3\rangle, \langle v_4\rangle\}]$.
5. $G :_1 [\{\} : \{\langle v_1\rangle, \langle v_2\rangle, \langle v_3\rangle, \langle v_4\rangle\}]$.
6. $B :_1 [\{\} : \{\langle v_1\rangle, \langle v_2\rangle, \langle v_3\rangle, \langle v_4\rangle\}$.
7. $no(R \cap G)$.
8. $no(R \cap B)$.
9. $no(G \cap B)$.
10. $(R \cup G \cup B) = V$.
11. $no((R \times R) \cap E)$.
12. $no((G \times G) \cap E)$.
13. $no((B \times B) \cap E)$.

The grounding approach used by Kodkod is based on the *Sparse-matrix representation of relations*. The idea is that a relation over a finite universe can be represented as a matrix of Boolean values. For example, a binary relation over universe $\{a_1, \cdots, a_n\}$ can be encoded with an $n \times n$ bit matrix that contains a 1 at the index $[i, j]$ when the relation includes the tuple $\langle a_i, a_j\rangle$. More generally, given a universe of n atoms, the collection of possible values for a relation variable $v :_k [l, u]$ (l is lower-bound and u is upper-bound from relation v) corresponds to a k-dimensional matrix M

$$M[i_1, \cdots, i_k] = \left\{ \begin{array}{ll} 1 & \text{if } \langle a_{i_1}, \cdots, a_{i_k}\rangle \in l \\ \nu(v, \langle a_{i_1}, \cdots, a_{i_k}\rangle) & \text{if } \langle a_{i_1}, \cdots, a_{i_k}\rangle \in u \backslash l \\ 0 & \text{if } \langle a_{i_1}, \cdots, a_{i_k}\rangle \notin u \end{array} \right\}, \tag{7.1}$$

where $i_1, \cdots, i_k \in \{1, \cdots, n\}$ and ν maps its inputs to unique Boolean variables. The matrix for non-atomic relations can be computed by the matrices of their components. The translation rules used by Kodkod are demonstrated in Appendix B.

Example 25 *The following shows how an answer to formula $no(R \cap G)$ can be computed in Kodkod:*

$$M_R = \begin{bmatrix} 1 \\ R_2 \\ R_3 \\ R_4 \end{bmatrix} \qquad M_B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix}$$

$$M_R \cap M_B = \begin{bmatrix} B_1 \\ R_2 \wedge B_2 \\ R_3 \wedge B_3 \\ R_4 \wedge B_4 \end{bmatrix} \qquad no(M_R \cap M_B) = \neg\left(B_1 \vee (R_2 \wedge B_2) \vee (R_3 \wedge B_3) \vee (B_4 \wedge R_4)\right)$$

Using suitable data structures for representing sparse matrices, each operation in Kodkod translation method (Appendix B) can be implemented such that the time complexity of computing the answer depends solely on the number of non-zero entries in the input matrices (but not the sizes of input matrices).

Pros and Cons

Although the input languages of KodKod and MXG are different, one can modify the algorithm used in each to work for the other input language. Essentially, a table (in MXG) can be seen as a data structure for representing a sparse matrix. Moreover every data structure for representing a sparse matrix can be used to represent an answer for a formula in MXG.

The main difference between MXG and Kodkod is that MXG produces a variable-free propositional formula first and then converts this formula to CNF, while Kodkod directly generates a SAT instance. As Kodkod generates CNF in a streaming manner, essentially, it assigns a CNF variable to each subformula. By this design, one can expect the number of variables in Kodkod's CNF be more than the CNF generated by MXG.

It is very straightforward to use KodKod to express the problems for which one knows some parts of the solution. Knowing that Kodkod performs well on problems whose corresponding matrices are sparse, there are some cases in which incorporating lower/upper bounds in the computation of matrices results in huge non-sparse matrices.

7.2 Top-Down Grounders

7.2.1 IDP

The input syntax of the IDP system [72] is very similar to that of the MXG system. IDP's grounding algorithm is a top-down grounder. The grounding algorithm used in IDP has two phases:

1. Computing Bounds,
2. Actual Grounding.

Given an instance structure $\mathcal{A}$ and a specification Φ, the IDP system tries to find two bounds, *certainly true-bound* (*ctb*) and *certainly false-bound* (*cfb*), for each subformula in the specification where each bound is a set of tuples. A tuple, $\bar{a}$, is in *ctb* of $\psi(\bar{x})$ (*cfb* of $\psi(\bar{x})$), ctb_ψ (cfb_ψ) iff $\psi(\bar{a})$ is true (false) in all models extending $\mathcal{A}$ and satisfying specification, Φ. More formally, for a subformula of ψ of Φ:

1. Tuple $\bar{a}$ is in ctb_ψ iff for every structure $\mathcal{B}$ that is an expansion of structure $\mathcal{A}$ to vocab of Φ and satisfies Φ we have
$$\mathcal{B} \models \psi(\bar{a}).$$
2. Tuple $\bar{a}$ is in cfb_ψ iff for every structure $\mathcal{B}$ that is an expansion of structure $\mathcal{A}$ to vocab of Φ and satisfies Φ we have
$$\mathcal{B} \models \neg\psi(\bar{a}).$$

After the bound computation phase, the system enters the grounding phase in which a ground formula, Ψ, is produced for the given specification, Φ, such that for every structure $\mathcal{B}$ which is an

expansion of structure $\mathcal{A}$ to vocab of Φ, $\mathcal{B} \models \Phi \Leftrightarrow \mathcal{B} \models \Psi$. Finally, the ground formula is fed into a SAT solver which has a richer syntax than CNF (it supports sets and aggregates).

7.2.2 Computing Bounds

There are two general approaches for computing *ctb* and *cfb* for subformulas. One is to use the instance structure information and the other one is to describe the *ctb* (*cfb*) of each subformula, symbolically, as a first-order formula of the instance predicate. The main benefit of the latter approach is:

There are specifications for which all ct and cf bounds have a short symbolic representation (there are short formulas describing *cfb* and *ctb* for all subformulas). As the formulas describing *cfb* and *ctb* do not change according to the value of instance predicates, this approach can be used for specifications with large domains.

The symbolic approach is used in the IDP system to compute the bounds. The specifications are rewritten as a collection of Implicational Normal Form (INF) formulas. A sentence is in INF if it is of the form $\forall \bar{x}\ \phi(\bar{x}) \rightarrow L[\bar{x}]$ where ϕ is an arbitrary formula and $L[\bar{x}]$ is a literal.

The IDP symbolic bound computation system computes bounds in two phases. In the first phase (the first loop in the algorithm illustrated in appendix C), a fix-point program is built based on the problem description (which is already transformed into conjunction of INF formulas). This fix-point program describes the *ctb* and *cfb* of a formula based on the bounds of the subformulas of that formula and the instance predicates. After constructing the program, the solving phase starts, in which IDP finds the least fix-point for the program.

Example 26 *Considering the specification presented in example 1, one can verify the following:*

- $R_p \subseteq ctb(R(x))$,
- $R_p \subseteq ctb(R(x) \vee B(x) \vee G(x))$,
- $R_p \subseteq cfb(B(x))$,
- $E(x,y) \vee E(y,x) \subseteq cfb(R(x) \wedge R(y))$.

To compute the fix-point correctly, one needs to have access to a function which can always detect whether the two first-order formulas are equivalent or not. We know that this problem, in its general case, is undecidable.

IDP's bound system uses the following technique. A canonical format for representing the formulas has been used in the system. The formulas are stored in a first-order BDD structure. So, if two formulas have the same canonical representation, it is guaranteed that those formulas are equivalent. In his thesis [72], Wittocx claims that a first-order BDD with 12 decision nodes is enough to handle most real-world specifications.

It can be shown that the above fixpoint program always has a fixpoint solution when the domains of variables are finite. But IDP's bound system may fail to find the solution because the computing fixpoint will terminate after a limited time. If this occurs, the bounds passed to the grounding phase will not complete.

7.2.3 Grounding Using Bounds

A grounding of a FO formula over a fixed domain A can be produced by recursively replacing each subformula in the form $\exists x\psi(x)$ with $\bigvee_{d\in A}\psi(\hat{d})$, where $\hat{d}$ is a (possibly new) constant symbol which is used to denote the domain object d, and dually replacing each subformula of the form $\forall x\psi(x)$ with a conjunction. For each fixed FO formula ϕ, this algorithm runs in time that is polynomial in $|A|$.

Having access to bounds for all subformulas, the naive algorithm can be optimized by avoiding some of the recursive replacing. As, if a pair of subformula and an assignment is in the bounds, the subformula must be true/false under that specific assignment. The algorithm in IDP uses the same logic (see Appendix D). By setting both bounds, certainly true-bound and certainly false-bound, to be empty sets, we achieve the naive top-down grounding.

The computed bounds are incomplete iff there are a subformula $\phi(\bar{x})$ and an instantiation $\bar{a}$ for $\bar{x}$ s.t., one can show that the value of $\phi(\bar{x}/\bar{a}) = \top(\bot)$ and $I \not\models C^{ctb}(\phi)[\bar{x}/\bar{x}]$ ($I \not\models C^{cfb}(\phi)[\bar{x}/\bar{x}]$, respectively). If the computed bounds are incomplete, the procedure still produces a correct grounding but in most cases, the resulting ground formula is large.

7.2.4 Pros and Cons

Performance of the Symbolic Bound computation procedure in IDP system is independent of the size of variables' domains. But this alone is not an important factor on the performance of the whole system because the running time of the grounding system in IDP depends on the size of the variables' domains.

Performance of the IDP grounder deeply depends on how accurate the computed bounds are. One can construct specifications for which IDP's bound system fails to find complete bounds.

The propagation rules for aggregates and functions are not complete; there are many cases which are ignored. To describe the ignored rules, one has to use a lengthy formula which in some cases cannot be represented using a first-order BDD with 12 nodes. Wittocx [72] claims that the situations in which the ignored rules are satisfied are not very common, and so the computed bounds are accurate enough.

Besides the fact that the fixpoint program, used in IDP to compute bounds, is not intended to find the best possible set of bounds, IDP's bound computation system may fail to detect that its current partial solution is actually a valid solution for the fix-point program, because all the computations are performed in a symbolic way. In the latter case, the system continues to enlarge the size of representation of bounds until it reaches the time limit.

To handle functions, the graph of a function is used in IDP. Every function, $f(\bar{x})$, is replaced by a predicate $G_f(\bar{x}, y)$ such that $y = f(\bar{x}) \Leftrightarrow G_f(\bar{x}, y)$ and then the atomic formulas are rewritten accordingly. This approach is not the best way to handle the function, as in certain cases, it makes the ground formula much bigger.

Another potential disadvantage of the IDP system comes from its implementation. The IDP system translates the resulting ground formula to an intermediate format which is an expansion of CNF. So, it cannot use the normal SAT solvers but instead must use SAT solvers which are able to handle their specific syntax. If a new SAT solver technique has been introduced, one must wait for IDP developers to extend that solver to work with the IDP intermediate language.

The advantage of using a symbolic representation for bounds is that the bounds are represented using a small data structure (comparing to the other option which is listing all the tuples in bounds). The disadvantage of the symbolic representation of bounds is the fact that one needs to compute the value of a formula (bounds) for a given instance structure, to check whether a tuple is within the bound or not.

7.3 Incremental Grounders

In this approach, a grounder selects a part of a given specification, grounds it and passes the ground formula to a low-level solver. If the low-level solver claims that there is no solution for the partially grounded specification, the grounder claims that the main problem does not have any solution. In the case when the low-level solver finds a solution, the grounder checks whether the obtained solution satisfies the whole specification or not.

If the obtained solution for the selected part is also a solution for the whole specification, the grounder reports that solution. If this is not the case, the grounder tries to find a (minimal) set of facts which makes the solution invalid and pass them back to the grounder. This process continues until a solution has been found or eventually the low-level solver claims that the ground formula does not have any solution.

While this idea is a well-studied approach in the ILP community, there is not any well-known solver which uses this approach. The main reason for this is that the hard problems have very few answers, while removing some of the constraints makes the problem much easier.

In the next subsection, we study a grounder for Answer Set programs.

7.3.1 DLV

DLV [49] is an Answer Set Programming (ASP, [15]) solver. Like many other ASP solvers, DLV computes a *ground program* from an instance of a given ASP program. In the ASP context, an ASP program which contains no variables is called the ground program.

Calimeri et al., in [21], extended the standard ASP language with complex terms:

- A term is either *simple term* or a *complex term*.
- A constant or a variable is a *simple term*.
- A *functional term* is defined as $f(t_1, \cdots, t_n)$, where f is a function symbol of arity n and $t_1, \cdots, t_n$ are terms.
- A *list term* can take the following two forms:
  1. $[t_1, \cdots, t_n]$, where $t_1, \cdots, t_n$ are terms;
  2. $[h|t]$, where h (the head of the list) is a term and t (the tail of the list) is a list term.
- A *set term* is defined as $\{t_1, \cdots, t_n\}$, where $t_1, \cdots, t_n$ do not contain any variables.

In [21], the authors describe some interesting problems which can be easily described in this enhanced language.

The idea used in the DLV-Complex to produce a ground program is to rewrite the complex term as an application of a predicate on the term's arguments. DLV-Complex assumes that the input ASP program is *safe*, meaning that all rules in that program are safe. A rule is safe when if a variable

appears in the head of a rule it also appears in the body of the same rule as an argument to a positive literal.

Under the safety assumption, the set of feasible values for every variable is finite. The DLV-Complex replaces any term $t = f(X_1, \cdots, X_n)$ appearing in some rule r by a fresh variable F; and then adds one of the two following atoms to body of r ($B(r)$):

- #-function-pack $(f, X_1, \cdots, X_n, F)$ if t appears in the head of r;
- #-function-unpack $(F, f, X_1, \cdots, X_n)$ if t appears in the body of r.

Intuitively, #-function-pack is in charge of building a function term, starting from a function symbol and arguments. #-function-unpack unfolds a functional term and extracts its function symbol and its arguments. The same procedure can be applied for more complex terms, too.

Example 27 *([21]) The rule* $p(f(f(X))) : -q(X, g(X, Y))$. *will be rewritten as follows:*

$$p(F_1) : -\#\textit{function-pack}(F_1, f, F_2), \#\textit{function-pack}(F_2, f, X), q(X, F_3), \\ \#\textit{function-unpack}(F_3, g, X, Y).$$

As the evaluation of a pure ASP program, an ASP program without complex terms, is not guaranteed to terminate in general [28]; the rewritten program is passed to a module, called Finite Domain Checker, which checks the membership to the finite-domain programs (the programs in this class are known to be computable) [22]. Finally, the rewritten program is given to DLV, an ASP solver used as the low-level solver in DLV-Complex [21].

One may notice that the approach IDP uses to ground functions, using graphs of functions, is similar to the idea used in DLV-Complex.

Pros and Cons

There are programs whose evaluations terminate but their rewritten programs, which are obtained by the DLV-Complex rewriting module, do not belong to the class of finite-domain programs. In a recent paper [52], it was shown that there are two equivalent programs, such that the DLV-Complex can ground one but not the other. This issue arises because the set of all finite-domain programs is not equal to the set of all ASP programs whose evaluations terminate.

It has been shown that the set of all programs which can be efficiently instantiated[3] is undecidable [22]; therefore it is impossible to have either a general syntactic or algorithmic approach to describe that class. Trying to express a larger set of finitely-ground programs is an active research area in the logic programming community. The class of argument-restricted programs which is a superset of the class of finite-domain programs, has been introduced [51]. This class also contains the ω-restricted programs [63], and λ-restricted programs [42].

Theoretically, the DLV-Complex can be extended to work if the rewritten program belongs to each of the above mentioned classes. But to be able to use DLV-Complex in practice, verifying whether an ASP program (the rewritten program) is in a certain class should be fast (checking whether a program is a finite-domain program is in linear time).

[3] Such programs are called finitely-ground programs

Another way to handle these cases is to modify the rewriting module. Given a program to the rewriting module, the rewriting module can automatically generate some of the programs equivalent to the input program. In this way, one can increase the chance of obtaining a ground program, a finite-domain program, for a given specification.

7.3.2 Other Grounding Techniques

Disjunctive Logic Programming (DLP) is an extension of Datalog that allows for disjunction in rule head and non-monotonic negation in bodies. This logic captures Σ^2 [35].

In [11], the authors proposed a grounder which reduces a program written in DLP enriched with monotone recursive aggregates to a normal DLP program. The idea of their grounding approach is to use the *Dynamic Magic Set technique*. The Magic Set method is a well-known approach in database optimization [14].

Roughly speaking, the method involves simulating query evaluations in a top-down manner and adding a set of rules to the original program in order to restrict the computation to a finite domain.

When discussing specifications with infinite domains, ILP solvers[4] are thought to be to be a good option. There are several ideas for using an ILP solver as the low-level solver in a model-based solver. One of the good features of these solvers is that expressing arithmetical terms in a linear program is usually straightforward. On the other hand, translating disjunctions in (Integer) linear programs is not straightforward at all.

[4]Integer Linear Program Solvers

Chapter 8

MakeCNF Phase

We described how Enfragmo obtains a variable-free FO formula from a given problem specification and problem instance. In order to use a SAT solver as Enfragmo's ground solver engine, we must develop modules which are able to translate a variable-free FO formula to an equi-satisfiable SAT instance.

In this chapter, we describe the different approaches we developed to create a SAT-instance, in the form of a CNF formula, from a given variable-free FO formula. We also describe how the place holders introduced in Sections 6.5 and 6.6 can be translated into CNF.

8.1 Introduction

The most common input format accepted by SAT solvers is Conjunctive Normal Form (CNF). Given a set of Boolean variables, $X = \{x_1, \cdots, x_n\}$, literal l is either a Boolean variable or its negation. A formula is in CNF if it is expressed as conjunction of clauses, where each clause is a disjunction of literals.

According to the Cook-Levin theorem [24] and the fact that any instance of Boolean satisfiability can be reduced to an equivalent 3SAT instance, we know that every instance of an NP problem can be reduced to a SAT instance, expressed in CNF. Sometimes, due to the abstract syntax accepted by SAT solvers, expressing a problem instance for these solvers is not straightforward. In the SAT community, one commonly used technique to reduce problems related to SAT is to construct a family of verifier Boolean circuits such that (1) the n-th circuit decides the membership of inputs of length n; (2) the size of the n-th circuit, which is the number of gates and wires in the circuit, is polynomial with respect to n. Each of the Boolean circuits receives a representation of the problem instance along with a witness as its input, and outputs One/*true* iff the provided witness is a proof for satisfiability of the instance.

Note that there may be several different CNF expressions for the same Boolean circuit. Also, different SAT solvers perform differently on the same CNF [76]. But generally speaking, for a given problem instance, the CNF expression with fewer variables is easier to solve. The length of clauses is another factor that affects the running time of SAT solvers. A SAT instance with short clauses is more likely to be solved than another instance with long clauses. Where there are two

CNF expressions for the same problem instance, the one that allows unit propagation to infer more will be easier to solve.

In the standard approach to translating a Boolean circuit to CNF, Tseitin transformation, [67], a fresh Boolean variable, is introduced for the output of every gate in the circuit, and the relation between the input(s) and output of each gate is described using a set of clauses.

In the Tseitin transformation, both the running time and the size of the resulting CNF formula are linear with respect to the number of gates in the input Boolean circuit. There might be many redundant/unnecessary variables and clauses in the resulting CNF expression. In this chapter, we describe an approach to produce CNF expressions which are easier to solve using SAT solvers.

In Sections 6.5 and 6.6, we described how an answer to a term can be computed using MIN, MAX, COUNT and SUM formulas (place holders). In this chapter, we explain how the first two formulas can be translated into the CNF formula. Different methods for translating an instance of COUNT formula (SUM formula) into CNF formula are discussed in Chapter 9 (10, respectively).

8.1.1 My Contribution

CNF Generator with "Fill and Return", described in Section 8.4, is contributed by Shahab Tasharrofi and the author. The methods, explained in Section 8.5, for translating place holders, $COUNT$, MIN, MAX, $BMIN$, $BMAX$, $BSUM$, and etc are proposed by the author. Postponing CNF generation for place holders, Section 8.6, is another contribution of the author.

All techniques, described in this chapter, have been implemented, by David Bergman and the author.

8.2 Background

In this section, we fix our notation and use them through the rest of this chapter. We also define what we mean by a *transformation*.

8.2.1 Notations

Let X be a set of Boolean variables. A literal, l, is either a Boolean variable or negation of a Boolean variable, and var (l) denotes the variable corresponding to l. A clause on X, $C = \{l_1, \cdots, l_m\}$, is a set of literals such that var $(l_i) \in X$. An assignment τ to X is a partial function that maps some variables in X to either *true* or *false*. By $x \in \tau^+ (x \in \tau^-)$ we mean that *true* (*false*) is assigned to x under assignment τ. Also, we use τ $[S]$, $S \subseteq X$, as a shorthand for the assignment obtained by restricting the domain of τ to the variables in S.

Assignment τ to X is a *total assignment* if it assigns a value to each variable in X, i.e., $\tau^+ \cup \tau^- = X$. Assignment τ satisfies literal l, $\tau \models l$, if $l = x$ and $x \in \tau^+$ or $l = \neg x$ and $x \in \tau^-$. Assignment τ satisfies clause $C = \{l_1, \cdots, l_m\}$ if there exists at least one literal l_i such that $\tau \models l_i$. A total assignment falsifies clause C if it does not satisfy any of its literals. An assignment satisfies a set of clauses if it satisfies all the clauses in that set. We say assignment τ' extends assignment τ, $\tau' \supseteq \tau$, iff both $\tau'^+ \supseteq \tau^+$ and $\tau'^- \supseteq \tau^-$ hold.

In the rest of this thesis, we use γ to represent an assignment to first-order variables and τ to denote an assignment to propositional variables. As in the previous chapters, we use $\mathcal{A}$ to represent a finite structure, A to denote the domain of $\mathcal{A}$. We assume $A = \{a_1, \cdots, a_n\}$.

Let Φ be a variable-free FO formula over vocabulary $\sigma \cup \varepsilon$, and $\mathcal{A}$ be a σ-structure. In this chapter, our aim is to develop algorithms to generate a CNF formula which is satisfiable iff there is a $\sigma \cup \varepsilon$-structure $\mathcal{B}$, expanding $\mathcal{A}$, such that $\mathcal{B} \models \Phi$.

Definition 27 *Let Φ be a variable-free FO formula over vocabulary $\sigma \cup \varepsilon$, and $\mathcal{A}$ be a σ-structure. We define a* transformation *to be a Boolean variable and a set of clauses, $\langle v, C\rangle$, such that:*

1. *C is satisfiable,*
2. *$C \cup \{v\}$ is satisfiable iff there is a structure $\mathcal{B}$, expanding $\mathcal{A}$, and $\mathcal{B} \models \Phi$.*

A transformation is the last component we need to describe in order to complete the reduction of an instance of model expansion to an instance of a satisfiability problem, expressed in CNF, and therefore solvable by SAT solvers.

8.2.2 Tseitin Transformation

Here, we briefly describe the Tseitin transformation [67], the standard method for transforming variable-free first-order formulas to CNF. In this transformation, a fresh propositional variable is created to represent the truth value of each subformula of the given formula. Let ψ_1, ψ_2, ψ be three such subformulas and x, y, z be the associated propositional variables to ψ_1, ψ_2 and ψ, respectively. The transformation works as follows:

1. $\psi = \psi_1 \vee \psi_2$: produce the following three clauses $\{z, \neg x\}, \{z, \neg y\}, \{\neg z, x, y\}$,
2. $\psi = \psi_1 \wedge \psi_2$: produce the following three clauses $\{\neg z, x\}, \{\neg z, y\}, \{z, \neg x, \neg y\}$,
3. $\psi = \neg\psi_1$: produce the following two clauses $\{\neg z, \neg x\}, \{z, x\}$,
4. $P(a_1, \cdots, a_n)$ is an atomic formula, where each $a_i \in A$: If it is the first time we encounter $P(a_1, \cdots, a_n)$, we introduce new propositional variable x and store $\langle x, P(a_1, \cdots, a_n)\rangle$, in an appropriate data structure. If we have already encountered $P(a_1, \cdots, a_n)$, we return the first argument of the pair stored in the memory.

Example 28 *Let $A = \{v_1, v_2\}$ the domain of $\mathcal{A}$, and $\phi = ((R(v_1) \vee G(v_1)) \vee B(v_1)) \wedge (\neg R(v_1) \vee \neg R(v_2))$ be a sentence. The following is a valid transformation for ϕ:*

Let X_1 be $\{r_1, g_1, b_1, r_2\}$, where r_1 represents the truth value of $R(v_1)$, g_1 represents the truth value of $G(v_1)$, and so on. Let o_1, o_2, o_3 and o_4 be four propositional variables corresponding to $R(v_1) \vee G(v_1)$, $(R(v_1) \vee G(v_1)) \vee B(v_1)$, $\neg R(v_1) \vee \neg R(v_2)$ and ϕ, respectively. The Tseitin transformation creates the following set of clauses to describe the truth value of $o_1, \cdots, o_4$:

- *$o_1 = (r_1 \vee g_1)$: $\{\neg r_1, o_1\}$, $\{\neg g_1, o_1\}$, and $\{\neg o_1, r_1, g_1\}$.*
- *$o_2 = (o_1 \vee b_1)$: $\{\neg o_1, o_2\}$, $\{\neg b_1, o_2\}$, and $\{\neg o_2, o_1, b_1\}$.*
- *$o_3 = (\neg r_1 \vee \neg r_2)$: $\{r_1, o_1\}$, and $\{r_2, o_1\}$, $\{\neg o_3, \neg r_1, \neg r_2\}$.*
- *$o_4 = (o_2 \wedge o_3)$: $\{o_2, \neg o_4\}$, $\{o_4, \neg o_4\}$, and $\{\neg o_2, \neg o_3, o_4\}$.*

In this example, tuple $o_2, \{\{\neg r_1, o_1\}, \{\neg g_1, o_1\}, \{\neg o_1, r_1, g_1\}, \{\neg o_1, o_2\}, \{\neg b_1, o_2\}, \{\neg o_2, o_1, b_1\}\}$ is a transformation for $(R(v_1) \vee G(v_1)) \vee B(v_1)$.

After computing a ground formula, which is a FO sentence, from the given specification and problem instance, Enfragmo translates the ground formula to a SAT instance. The main argument against using the Tseitin transformation is that there are problems whose corresponding CNF, generated using the Tseitin transformation, cannot be solved efficiently. The author of this thesis detected the following issues as the main shortcomings of the Tseitin transformation:

1. The generated CNF is not compact: The CNF contains clauses which can be eliminated without affecting the satisfiability of the problem,
2. There are many redundant variables in the CNF: For certain problems, some of the variables in the generated CNF can be eliminated,
3. There are several equivalent variables in the CNF: Depending on the problem, some of the Tseitin variables are used to represent the same formula, therefore they can be merged/eliminated.

Een detected the same issues, and proposed solutions to achieve easier to solve CNFs [33]. In Section 8.3, we discuss some further modifications, proposed by the author of this thesis, to the Tseitin transformation to address the above mentioned issues. In Section 8.4, we describe an algorithm to construct a transformation from a variable-free FO sentences which is, to the best of author's knowledge, has not been proposed elsewhere.

8.3 Improving The Tseitin Transformation

In this section, we describe some modifications to the Tseitin transformation to its performance.

8.3.1 Huge CNF

Intuitively, having many clauses reduces the search space, but increases the running time of the Make CNF phase and also the running time of the unit propagation module in the SAT solver. Therefore, it is desirable to reduce the number of clauses without affecting the performance of unit propagation. The following two observations can be used to produce a smaller CNF:

- If the ground formula obtained as the result of the grounding phase has an "AND" node as its root node, we do not need to create a Tseitin variable for that node, since the ground formula can be considered as the conjunction of two (or more) smaller formulas. We can then translate each of those formulas to SAT separately. The same is true for "AND" nodes whose ancestors are also "AND" nodes.
- Sometimes, the CNF generator module creates some unary clauses, and so the value of some variables would be forced. These variables may occur in some other clauses too.

There are techniques in the literature for modifying the Tseitin transformation to address the above observations [33].

8.3.2 Redundant Auxiliary the Tseitin Variables

The number of Boolean variables in a SAT instance may have a huge impact on the running time of SAT solvers. Usually, the fewer the number of variables, the smaller the search space is. We

observed that the following situations cause the Tseitin transformation to create redundant auxiliary variables:

- If the same Boolean subformula appears in several places of the ground formula: The naive method for generating CNF uses different Tseitin variables to describe each occurrence of that subformula.
- If the ground formula has several layers of consecutive AND nodes: These AND nodes can be merged together and form a single AND node (which has more children than each of the original AND nodes). By merging the consecutive AND nodes (consecutive OR nodes), we can reduce both the number of variables and the number of clauses.

Methods Used in Enfragmo

Here we briefly explain how we modified the Tseitin transformation in Enfragmo. These modifications can be used in any system.

Memorization To detect multiple occurrences of a formula, the CNF generator method in Enfragmo has been equipped with a memory (implemented as a hash map data structure). Whenever the CNF generator method needs to create a Tseitin variable for a subformula, at first, it searches the memory. If the subformula is not found, a new variable will be created and the pair of variable and subformula will be stored in the memory. Otherwise, the CNF generator method uses the existing variable in the memory.

For certain problems, using this approach is not useful since the ground formula does not contain many repeated subformulas and the search overhead increases the running time of the MakeCNF phase drastically. One can name the ground formula obtained from the naive encoding of a graph colouring problem as an example of such a case.

At present, Enfragmo allows users to turn on/off the memorization option in the CNF generator method. To automate this decision, we can use the following heuristic.

It is beneficial to use memorization if there are many occurrences of the same subformulas in the ground formula. So, we can sample a subset of subformulas in the produced CNF and count how many are the same. If the size of the sample set is big enough, the number of similar formulas in the sample set can be used to estimate how much we can save by turning on the memorization switch.

Merging "AND" nodes Een proposed to create no Tseitin variable for a node if all its parents nodes are also AND nodes [33]. As we explain in the next section, Enfragmo uses the FillAndReturn algorithm. This algorithm extends the idea proposed in [33], and merges consecutive "AND" nodes ("OR" nodes).

The CNF generator algorithm used in Enfragmo merges the consecutive AND nodes (OR nodes). It avoids creating a Tseitin variable for NOT nodes (if the variable describing the node beneath a NOT node is v, then the literal $\neg v$ can be used as a representative for that NOT node). This idea allows us to rewrite a NOT node whose child is an AND node as an OR node. Similarly, we can rewrite a NOT node whose child is an OR node as an AND node.

8.4 CNF Generation with Fill and Return

In this section, we describe a new approach for generating CNF from a variable-free FO formula. This approach, FillAndReturn, is a generalization of the improvements proposed in [33].

Before we describe the FillAndReturn algorithm, we believe it is useful to discuss the data structure used in Enfragmo to represent formulas. In Enfragmo, a formula is represented using a Directed Acyclic Graph (DAG), where each internal node of the graph corresponds to an operator in the formula, and leaf nodes of the graph are atomic formulas. To represent formulas compactly, we introduce a special internal node, assignment node, which assigns a value to a variable.

The input to FillAndReturn algorithm is a formula, represented as a DAG. There are four main methods in this algorithm: PositiveFill, PositiveReturn, NegativeFill and NegativeReturn. The return methods (PositiveReturn and NegativeReturn) take a formula node, R_f, and return a Tseitin variable corresponding to R_f. The Fill methods, PositiveFill and NegativeFill, take a formula node, R_f, along with an array A, and fill array A, by collecting the Tseitin variables of children of R_f. We say the node N and its descendant node C have the same type if:

1. Both N and C are AND nodes and C is a direct child of N.
2. Both N and C are AND nodes, C is not a direct child of N and on the path P connecting N to C, all nodes are either NOT nodes or Assignment nodes, and P has an even number of NOT nodes.
3. N is an AND node while C is an OR node, C is not a direct child of N and on the path P connecting N to C, all nodes are either NOT nodes or Assignment nodes and P has an odd number of NOT nodes in it.
4. Both N and C are OR nodes and C is direct child of N.
5. Both N and C are OR nodes, C is not a direct child of N and on the path P connecting N to C, all nodes are either NOT nodes or Assignment nodes, and P has an even number of NOT nodes.
6. N is an OR node while C is an AND node, C is not a direct child of N and on the path P connecting N to C all nodes are either NOT nodes or Assignment node and P has an odd number of NOT nodes in it.

Given a formula R_f, we say node N is in a positive position (negative position) if there an even (odd) number of NOT nodes on the path connecting N to the root of R_f. PositiveFill (NegativeFill) is called for the nodes that are in positive position (negative position, respectively) and collects all the Tseitin variables in an array data structure. If a child has the same type as its parent, the fill function visits the child's children and adds them to the array, otherwise, a Tseitin variable is created for the child node, by calling the appropriate return method.

Algorithms 8, 9 and 10 illustrate different functions in the FillAndReturn algorithm. We have only described how PositiveFill and PositiveReturn work. The two other methods, NegativeFill and NegativeReturn, can be defined, dually. The methods for translating atomic formulas to CNF, e.g., EncodePredicate, EncodeCOUNT, EncodeMax, EncodeMin and EncodeSum, are explained in the next section.

Algorithm 8 POSITIVEFILL method takes $Root$ and $Assignment$ as its input, and fills $ANARRAY$ with Tseitin variables.

Input: A Formula $Root$, an array of Boolean literals $AnArray$, Current Assignment $Assignment$.
Output: $AnArray$ contain a Tseitin variable for each of its descendant, with whom $Root$ has the same type.

```
1:  function POSITIVEFILL(Root, AnArray, Assignment)
2:      NegationCounter= 0;
3:      for C such that C is a child of Root do
4:          while C is a NOT node or C is an Assignment node do
5:              if C is a NOT node then
6:                  NegationCounter++
7:                  C= C.GetChild ()
8:              else
9:                  Import Assignment into Assignment
10:                 C= C.GetChild ()
11:         if C is an atomic formula then
12:             if Even (NegationCounter) then
13:                 AnArray.Insert (MakeCNFForAtomicFormula(C,Assignment));
14:             else
15:                 AnArray.Insert (Negate (MakeCNFForAtomicFormula(C,Assignment)));
16:         else if C and Root have the same type then
17:             if Even (NegationCounter) then
18:                 PositiveFill (C, AnArray, Assignment);
19:             else
20:                 AnArray.Insert (NegativeReturn (C, Assignment));
21:         else{ C and Root have different types.}
22:             if Even (NegationCounter) then
23:                 AnArray.Insert (PositiveReturn (C, Assignment));
24:             else
25:                 NegativeFill (C, AnArray, Assignment);
```

The POSITIVEFILL method, given formula node $Root$ and a partial assignment, visits all nodes beneath $Root$ which have the same type as $Root$. POSITIVEFILL calls NegativeReturn for the descendant nodes of $Root$ which do not have the same type as $Root$. As we explained, this method fills $AnArray$ with appropriate literals.

Algorithm 9 NEGATIVERETURN method takes $Root$ and $Assignment$ as its input, and it returns a Boolean variables corresponding to $Root$, under assignment $Assignment$.

Input: A Formula $Root$, Current Assignment $Assignment$
Output: Returns a Tseitin variable corresponding to formula $Root$

```
1:  function POSITIVERETURN(Root, Assignment)
2:      NegationCounter = 0, Array = ∅
3:      for C such that C is a child of Root do
4:          while C is a NOT node or C is an Assignment node do
5:              if C is a NOT node then
6:                  NegationCounter++;
7:                  C= C.GetChild ();
8:              else
9:                  Import Assignment into Assignment
10:                 C= C.GetChild ();
11:         if C is an atomic formula then
12:             if Even (NegationCounter) then
13:                 Array.Insert (MakeCNFForAtomicFormula(C,Assignment));
14:             else
15:                 Array.Insert (Negate (MakeCNFForAtomicFormula(C,Assignment)));
16:         else if C and Root have the same type then
17:             if Even (NegationCounter) then
18:                 PositiveFill (C, Array, Assignment);
19:             else
20:                 Array.Insert (NegativeReturn(Ch, Assignment));
21:         else                                   ▷ C and Root have different types.
22:             if Even(NegationCounter) then
23:                 Array.Insert (PositiveReturn (C, Assignment));
24:             else
25:                 NegativeFill (C, AnArray, Assignment);
26:     Result= EncodeAtomic (Array, Root.GetOperation ());
27:     return Result;
```

The NEGATIVERETURN method, given formula node $Root$ and a partial assignment, generates a fresh Boolean variable by calling ENCODEATOMIC method.

Algorithm 10 ENCODEATOMIC method takes $Array$ and $FormulaType$ as its input, and returns a Boolean literal.

Input: An array of Boolean literals $Array$, $FormulaType$ which is one of INST, EXP, COMP, AND, OR, COUNT, MAX, MIN, SUM.

Output: Returns a Boolean literal for the result of applying operator $FormulaType$ on the Boolean literals in $Array$.

```
1:  function EncodeAtomic(Array, FormulaType)
2:      Result= a fresh Tseitin variable
3:      if FormulaType is AND then
4:          for l in Array do
5:              Add clause {¬Result, l}.
6:          Add clause {Result, l_1, ··· , l_k : l_i ∈ Array}.
7:      else if FormulaType is OR then
8:          for l in Array do
9:              Add clause {Result, l}.
10:         Add clause {¬Result, l_1, ··· , l_k : l_i ∈ Array}.
11:     else if FormulaType is COUNT then
12:         EncodeCOUNT (Array, n)
13:     else if FormulaType is Max then
14:         EncodeMax (Array, n)
15:     else if FormulaType is Min then
16:         EncodeMin (Array, n)
17:     else if FormulaType is Sum then
18:         EncodeSum (Array, n)
        return Result;
```

The EncodeAtomic method, given array of literals $Array$ and a formula type, creates fresh Boolean variable $Result$. This method, by using the appropriate set of clauses, relates the truth-value of Boolean variable $Result$ with the truth-value of the literals in $Array$.

It is worth mentioning that the FillAndReturn algorithm generates clauses in the PositiveReturn and NegativeReturn methods and also whenever it reaches a place holder. The two fill methods only collect the generated Tseitin variables.

8.5 CNF Generation for Atomic and Place Holder Formulas

In Section 8.3 and Section 8.4, we assumed that we know how to translate atomic formulas into CNF. In this section, we describe how we deal with atomic formulas.

In all of the grounding algorithms described in the previous chapters the terms in the final ground formulas are either variables or constants. We know there is no free variable in the result of the grounding phase. So, if there is a node in the DAG representation of the ground formula that corresponds to a variable, it is guaranteed that there is an Assignment node above it. The atomic subformulas of the final ground formula, obtained as the result of grounding, are one of the following (We

use γ as the assignment described by $Assignment$ variable in Algorithms 8 and 9).

1. Instance Predicate $P(x_1, \cdots, x_n)$: The atomic formula $P(x_1, \cdots, x_n)$ evaluates to *true* or *false*, depending on whether $\gamma \in P^{\mathcal{A}}$ or $\gamma \notin P^{\mathcal{A}}$.
2. Comparison between two variables, $v_1\{\leq, <, =, >, \geq\}v_2$: The atomic formula v_1 *op* v_2, where $op \in \{\leq, <, =, <, \geq\}$, evaluates to either *true* or *false*, depending on the values assigned to v_1 and v_2 by γ.
3. Expansion Predicate: If a Boolean variable is already created for $P[\gamma]$, that variable will be returned, otherwise a fresh Boolean variable will be created and returned.
4. MIN/MAX place holders: In Subsection 8.5.1, we describe how these formulas, introduced in Section 6.5, can be translated into CNF,
5. COUNT place holders: In Chapter 9, we describe how an occurrence of COUNT formula, introduced in Section 6.5, can be translated to a cardinality constraint and how a cardinality constraint can be expressed as a SAT instance,
6. SUM place holders: In Chapter 10, we describe how an occurrence of SUM formula, introduced in Section 6.6, can be translated to a Pseudo-Boolean constraint and how a Pseudo-Boolean constraint can be expressed as a SAT instance,
7. BMIN/BMAX/BEQ/BLESSEQ/BADD/BSUB/BMUL place holders: In Subsection 8.5.2, we describe how these formula, introduced in Section 6.6, can be translated into CNF.

Similar to the previous chapter, we assume all variables are of type T. Let $\phi(\bar{x})$ be an atomic formula, and γ be an assignment mapping every variable occurring in $\bar{x}$ to a member of T. It is clear how the first two types of atomic formulas can be translated to CNF, as they can be evaluated and replaced with *true* or *false*. To translate a formula of the third type, i.e., expansion predicates, Enfragmo is equipped with the following two techniques.

- *Static Variable Generator:* Let the given specification list $P_1, \cdots, P_n \in \varepsilon$ as the expansion predicates, and also let d_i be the arity of predicate P_i, (assuming we just have one sort). We pre-allocate the first $|T|^{d_1}$ Boolean variables for P_1, the next $|T|^{d_2}$ Boolean variables for P_2, and so on.
  Given assignment γ, let $\langle v_1, \cdots, v_m\rangle$ be the values assigned to $\langle x_1, \cdots, x_m\rangle$. We define the rank of, $Rank(\langle v_1, \cdot, v_m\rangle)$, to be the number of tuples in T^m which are less than $\langle v_1, \cdots, v_m\rangle$. Then Enfragmo uses the Boolean variable with the following index to specify the truth-value of atomic formula $P_i[\gamma]$:

$$Rank(\langle v_1, \cdots, v_m\rangle) + \sum_{j=1}^{i-1} |T|^{d_j} \quad .$$

  The value of $Rank(\langle v_1, \cdots, v_m\rangle)$, where $T = \{t_1, \cdots, t_n\}$, can be computed in linear time with respect to m using the following formula (if T is not an ordered set, we can define an arbitrary ordering on it).

$$Rank(\langle v_1, \cdots, v_m\rangle) = \sum_{i=1}^{m} |T|^{i-1} * rank(v_i),$$

  where $rank(v) = |\{x \mid x \in T, x < v\}|$, i.e., the number of elements in T which are less than v.

- *Dynamic Variable Generator:* In this approach, Enfragmo uses a look-up table.
  The look-up table used in Enfragmo is based on the red-black tree data structure which maps the pair of integers, $\langle k, Rank\rangle$, to a Boolean variable. Given ground atomic formula $P_k[\gamma]$, where P_k is an expansion symbol, Enfragmo computes tuple $\langle v_1, \cdots, v_m\rangle$ such that v_i is the value assigned to x_i by assignment γ. Then it checks if the look-up table contains an entry with key equal to pair $\langle k, Rank(\langle v_1, \cdots, v_m\rangle)$. If the look-up table contains such an entry, the corresponding Boolean variable will be returned. Otherwise, a fresh Boolean variable will be assigned to the pair $\langle k, Rank(\langle v_1, \cdots, v_m\rangle)\rangle$ and will be inserted into the look-up table.

The running time of the Static Variable Generator (SVG) is linear with respect to the arity of P_k, while the running time of Dynamic Variable Generator (DVG) is linear with respect to the arity of P_k and is logarithmic with respect to the number of entries in the red-black tree data structure. Therefore if there are many entries in our red-black tree data structure, DVG performs slower than SVG. On the other hand, for most of instances, the number of Boolean variables generated by SVG is much greater than the number of variables generated by DVG for the same input. Users can select whether to use SVG or DVG in the MakeCNF phase.

8.5.1 MakeCNF for MIN/MAX place holders

Here we focus on explaining how an instance of MIN place holder can be translated into SAT. The procedure for translating an instance of a MAX formula is similar.

In Chapter 6, we used $MIN_{\mathcal{S},\mathcal{R},d_m,n}[\gamma]$, where γ is an assignment mapping each variables occurring in $\bar{x}$ to an element in T, $\mathcal{S}$ is an answer to formula $\phi(\bar{x},\bar{y})$, $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}})$ is a unary answer to term $t(\bar{x},\bar{y})$, and both d_m and n are integers, as the place holder for the ground formula corresponding to the following formula:

$$(Min_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m\} = n)[\gamma]. \tag{8.1}$$

Recall that we use $\gamma \cup \gamma_i$, where γ and γ_i are two assignments, to describe the assignment obtained by merging two assignments γ and γ_i.

Let X be the set of variables occurring in $\bar{x}$. For every $\gamma_i : X \mapsto T$ and $o_i \in \alpha_{\mathcal{R}}$, formula $\delta_{\mathcal{S}}(\gamma_i \cup \gamma) \wedge \delta_{\beta_{\mathcal{R}}}(o_i, \gamma_i \cup \gamma)$ describes the conditions for satisfiability of the following expression:

$$(\phi(\bar{x},\bar{y}) \wedge (t(\bar{x},\bar{y}) = o_i))[\gamma \cup \gamma_i]. \tag{8.2}$$

Satisfiability of Formula 8.2 means that the value of $Min_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m\}$ is less than or equal to integer o_i. As the first step to computing a transformation for the MIN formula, we compute the set of the following pairs, $\langle v_i, c_i\rangle$, and denote the set of pairs by $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$:

For every $\gamma_i : X \mapsto T$ and $o_i \in \alpha_{\mathcal{R}}$, let propositional variable v_i be the first component of pair $\langle v_i, C_i\rangle$ where $\langle v_i, C_i\rangle$ is a transformation of formula $\delta_{\mathcal{S}}(\gamma_i \cup \gamma) \wedge \delta_{\beta_{\mathcal{R}}}(o_i, \gamma_i \cup \gamma)$.

From the above construction, having pair $\langle o_i, v_i\rangle$ in $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ implies that there is a satisfying assignment for C_i in which v_i is mapped to *true* iff there is structure $\mathcal{B}$ expanding $\mathcal{A}$ and assignment γ_i such that $\mathcal{B}$ models Formula 8.2.

Given total assignment τ, set $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ can be seen as a constraint (propositional constraint) which evaluates to *true* iff the minimum over all o_i, such that τ maps the corresponding v_i to *true*, is equal to n. In the other words, constraint $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ evaluates to true iff

1. There is at least one i, such that $o_i = n$ and v_i is mapped to *true*,
2. For all j where $o_j < n$, v_j is mapped to *false*.

Proposition 21 describes how we transform $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ to a CNF formula.

Proposition 21 *Let* $VL^{\gamma,n}_{\mathcal{R},\mathcal{S}} = \{v_i \mid \langle n, v_i\rangle \in MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}\}$ *and* $VL^{\gamma,n<}_{\mathcal{R},\mathcal{S}} = \{v_i \mid (o_i < n)$ *and* $\langle o_i, v_i\rangle \in MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}\}$. *Let* v_o *be a fresh Boolean variable and* C' *be the set of clauses expressing the following propositional formula:*

$$v_o \leftrightarrow \bigvee_{v \in VL^{\gamma,n}_{\mathcal{R},\mathcal{S}}} v \wedge \bigwedge_{v \in VL^{\gamma,n<}_{\mathcal{R},\mathcal{S}}} \neg v. \tag{8.3}$$

Then pair $\langle v_o, C' \cup \bigcup_i C_i\rangle$ *is a transformation for* $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ *and also for atomic formula* $MIN_{\mathcal{R},\mathcal{S},d_m,n}[\gamma]$.

Proof: Set $VL^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ contains all propositional variables v_i such that their corresponding o_i in $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ are equal to n. Similarly, set $VL^{\gamma,n<}_{\mathcal{R},\mathcal{S}}$ contains all v_i such that their corresponding o_i, in $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$, are less than n. Equation 8.3 enforces that v_o is mapped to *true* iff there exists at least one v in $VL^{\gamma,n}_{\mathcal{R},\mathcal{S}}$ such that v is mapped to *true*, and all $v \in VL^{\gamma,n<}_{\mathcal{R},\mathcal{S}}$ are mapped to *false*. Therefore, there exists assignment γ_i, such that $(\phi(\bar{x},\bar{y}) \wedge (t(\bar{x},\bar{y}) = n))[\gamma \cup \gamma_i]$ is *true*, and for all $\gamma_j \neq \gamma_i$, we have either $\neg\phi(\gamma_j|_{\bar{x}}, \gamma|_{\bar{y}})$ or $t(\gamma_j|_{\bar{x}}, \gamma|_{\bar{y}}) > n$. Therefore there is a satisfying assignment for the set of clauses $C' \cup \bigcup C_i$ such that it maps v_o to *true* iff the original MIN formula is satisfiable.

■

8.5.2 MakeCNF for Atomic Formulas Used in a Binary Answer to Term

In this subsection, we briefly describe how an atomic formula in form of $BEQ_{n,\mathcal{R}}(\gamma)$, $BLESSEQ_{\mathcal{R},\mathcal{S}}(\gamma)$, $BMIN_{\mathcal{T},\mathcal{R},n,d_M}(\gamma)$ or $BADD_{\mathcal{R},\mathcal{S},n}(\gamma)$ can be transformed to CNF. The transformation for the other atomic formulas used to describe binary answers to terms can be constructed similarly.

For the sake of explanation, we assume all the integers in the range of terms are positive, and so we ignore $\alpha_{\mathcal{R}}$ in the rest of this subsection. All the arguments and algorithms can be extended to the general case.

Before we explain the transformations, we need to introduce new notation.

Definition 28 *Let* $\mathcal{R} = (\alpha_{\mathcal{R}}, \beta_{\mathcal{R}}, m_{\mathcal{R}}, M_{\mathcal{R}})$ *be a binary answer to term* $t(\bar{x})$ *with respect to structure* $\mathcal{A}$. *We use* $L_{\mathcal{R}}$ *to denote the number of bits needed to represent the absolute values of integers in the range of* $t(\bar{x})$. *Choosing* $L_{\mathcal{R}}$ *to be* $\log(\max(|m_{\mathcal{R}}|, |M_{\mathcal{R}}|))$ *enables us to represent the absolute value of any integer between* $m_{\mathcal{R}}$ *to* $M_{\mathcal{R}}$.

We use $\mathcal{R}(\gamma)$, *where* γ *is an assignment to the variables occurring in* $\bar{x}$, *to denote the binary representation of the absolute value of* $t[\gamma]$, *i.e.,* $\langle \delta_{\beta_{\mathcal{R}}(L_{\mathcal{R}})}(\gamma)), \cdots, \delta_{\beta_{\mathcal{R}}(0)}(\gamma)\rangle$.

Since we assumed all the integers in the range to be positive, $\mathcal{R}(\gamma)$ represents the value of $t[\gamma]$, too.

Definition 29 *Let $\mathcal{R}$ be a binary answer to term $t(\bar{x})$. We define $B_{\mathcal{R}}$ to be the following set*

$B_{\mathcal{R}} = \{\langle \gamma, b_{L_{\mathcal{R}}}, \cdots, b_0\rangle : b_i$ *is the first element of transformation for* $\delta_{\beta_{\mathcal{R}}(i)}(\gamma)$, $i = 0, \cdots, L_{\mathcal{R}}\}$.

We use $B_{\mathcal{R}}(\gamma)$ to refer to the tuples $\langle b_{L_{\mathcal{R}}}, \cdots, b_0\rangle$ where $\langle \gamma, b_{L_{\mathcal{R}}}, \cdots, b_0\rangle \in B_{\mathcal{R}}$. Notice that $B_{\mathcal{R}}(\gamma)$ denotes the value of $t[\gamma]$, in the binary representation.

Transforming $BEQ_{n,\mathcal{R}}[\gamma]$ to CNF

The atomic formula $BEQ_{n,\mathcal{R}}(\gamma)$, where n is an integer and $\mathcal{R}$ is a binary answer to a term, evaluates to *true* iff $\mathcal{R}(\gamma)$ equals n. Since we assumed all integers are positive, to translate an occurrence of $BEQ_{n,\mathcal{R}}(\gamma)$ to CNF, we only need to compare the corresponding bits in the binary representation of n and $B_{\mathcal{R}}(\gamma)$.

Let $\langle n_l, \cdots, n_0\rangle$ be the binary representation of n and v be a fresh Boolean variable. Also let $\langle b_{L_{\mathcal{R}}}, \cdots, b_0\rangle$ be the tuple represented by $B_{\mathcal{R}}(\gamma)$. Then pair $\langle v, C\rangle$ is a transformation for $BEQ_{n,\mathcal{R}}(\gamma)$, where C is any set of clauses equivalent to the following propositional formula:

$$v \leftrightarrow \bigwedge_{\substack{0\le i\le L_{\mathcal{R}}\\ n_i \text{ is one}}} b_i \wedge \bigwedge_{\substack{0\le i\le L_{\mathcal{R}}\\ n_i \text{ is zero}}} \neg b_i.$$

Transforming $BLESSEQ_{\mathcal{R},\mathcal{S}}[\gamma]$ to CNF

Atomic formula $BLESSEQ_{\mathcal{R},\mathcal{S}}(\gamma)$, where both $\mathcal{R}$ and $\mathcal{S}$ are binary answers to terms, evaluates to *true* iff the integer represented by $\mathcal{R}(\gamma)$ is less than or equal to the integer represented by $\mathcal{S}(\gamma)$. Since we assumed all integers are positive, to translate an instance of $BLESSEQ_{\mathcal{R},\mathcal{S}}(\gamma)$ to CNF, we just need to compare $B_{\mathcal{R}}(\gamma)$ and $B_{\mathcal{S}}(\gamma)$.

Let B_1 and B_2 be two tuples corresponding to $B_{\mathcal{R}_1}(\gamma)$ and $B_{\mathcal{R}_2}(\gamma)$, respectively. and O be a fresh propositional variable. Let $CMPT(B_1, B_2)$ denote a set of clauses such that for every satisfying assignment τ for $CMPT(B_1, B_2)$, we have:

$\tau \models O$ iff the unsigned integer represented by B_1, under assignment τ, is less than or equal to the unsigned integer represented by B_2, under assignment τ.

This transformation is well-studied in the SAT community and we do not describe it here [33].

Transforming $BMIN_{\mathcal{S},\mathcal{R},n,d_m}[\gamma]$ to CNF

Let $t(\bar{y})$ be $Min_{\bar{x}}\{t_1(\bar{x}, \bar{y}) : \phi(\bar{x}, \bar{y}); d_m\}$. A $BMin$ place holder evaluates to *true*, iff the n-th bit in the result of the Min aggregate is one. The result of the Min aggregate can be computed using a well-known construct in the SAT community, called ITE gates.

In the context of this thesis, $ITE(v, \langle B_l, \cdots, B_0\rangle, \langle b_l, \cdots, b_0\rangle, \langle O_l, \cdots, O_0\rangle)$ is the CNF relating the truth value of O_i to the truth values of b_i and B_i, $0 \le i \le l$, such that, in all satisfying assignments for $ITE(v, \langle B_l, \cdots, B_0\rangle, \langle b_l, \cdots, b_0\rangle, \langle O_l, \cdots, O_0\rangle)$, we have

- O_i and B_i have the same truth value if v is *true*: This can be described using any set of clauses equivalent to: $v \wedge B_i \rightarrow O_i$ and $v \wedge \neg B_i \rightarrow \neg O_i$.

- O_i and b_i have the same truth value if v is *false*: This can be described using any set of clauses equivalent to: $\neg v \wedge b_i \rightarrow O_i$ and $\neg v \wedge \neg b_i \rightarrow \neg O_i$.

There are other approaches to constructing a transformation for an ITE gate [33].

Let $\gamma_1, \cdots, \gamma_{|T|^{|\bar{x}|}}$ be all possible assignments mapping each variable occurring in $\bar{x}$ to elements in T. Let $O^i = \langle O^i_l, \cdots, O^i_0 \rangle$ be a tuple of propositional, $i = 0, \cdots |T|^{|\bar{x}|} + 1$. Using ITE and $CMPT$ transformations, we can construct a set of clauses such that in all its satisfying assignments, we have:

1. The truth-value of O^i_j is the same as the truth values of the j-th propositional variable in tuple $B_{\mathcal{R}}(\gamma_i)$, if we have $\delta_{\mathcal{S}}(\gamma, \gamma_i)$ is *true*, and the integer $\mathcal{R}(\gamma_i)$ is less than or equal to the integer represented by O^{i-1}; for $1 \leq i \leq |T|^{|\bar{x}|}$ and $0 \leq j \leq l$.

2. The truth-value of O^i_j is the same as the truth values of the O^{i-1}_j, if either $\delta_{\mathcal{S}}(\gamma, \gamma_i)$ is *false*, or the integer $\mathcal{R}(\gamma_i)$ is less than or equal to the integer represented by O^{i-1}; for $1 \leq i \leq |T|^{|\bar{x}|}$ and $0 \leq j \leq l$.

We select the value of variables in O^0 such that the corresponding integer is greater than the maximum value $\mathcal{S}$ can take, e.g., $M_{\mathcal{S}} + 1$. At the last step, we set the truth-value of variables in $O^{|T|^{|\bar{x}|}+1}_j$ to be the same truth-value $O^{|T|^{|\bar{x}|}}_j$, if the integer represented by $O^{|T|^{|\bar{x}|}}$ is less than or equal to $M_{\mathcal{S}}$. Otherwise, we set the truth-value of propositional variables $O^{|T|^{|\bar{x}|}+1}_j$ such that the integer represented by $O^{|T|^{|\bar{x}|}+1}$ is equal to d_m.

Then the pair $\langle O^{|T|^{|\bar{x}|}+1}_n, C \rangle$ is a transformation for $BMIN_{\mathcal{T},\mathcal{R}_1,n,d_m}[\gamma]$, where C is the set of all clauses generated during this construction.

An occurrence of the $BMAX$ atomic formula can be translated to CNF in a similar way.

Transforming $BADD_{\mathcal{R},\mathcal{S},n}[\gamma]$ to CNF

An occurrence of the $BADD$ formula can be translated into CNF using a transformation for an adder circuit which adds $B_{\mathcal{R}}(\gamma)$ to $B_{\mathcal{S}}(\gamma)$. One simple way to construct an adder circuit is to use a series for full-adders [33].

To transform $BSUB$ and $BMUL$ to CNF, we can use subtracter and multiplier circuits.

8.6 Lazy CNF Generation for Terms

Consider the result of the grounding of the formula presented in Example 29.

Example 29 *Let us assume we have a specification where I is an instance predicate, f is an instance function, E is an expansion predicate and all variables are in the range of $\{1, \cdots, 100\}$:*

$$\exists y\, E(y) \wedge \exists x\, I(x) \wedge (x = Min_y\{f(y) : E(y); 100\})\,. \tag{8.4}$$

Assuming $E(y)$ describes a line-up of people, $E(y)$ is true iff person y is in the line-up, and $f(y)$ returns the age of person y. Then the above axiom asserts that the line-up is non-empty and

the age of the youngest person in the line-up must be in $I^{\mathcal{A}}$. The ground formula for axiom 8.4 has the following structure:

- *For each value of $x \in I^{\mathcal{A}}$, the ground formula has an occurrence of the MIN formula.*
- *For each MIN formula corresponding to $Min_y\{f(y) : E(y); 100\}$ aggregate, there are 100 tuples in $MC^{\gamma,n}_{\mathcal{R},\mathcal{S}}$, defined in Subsection 8.5.1.*
- *All of the sets $M^{\gamma,n}_{\mathcal{R},\mathcal{S}}$, for different value of n, have exactly the same content. They differ only in their corresponding output values, n.*

Let's set the interpretation for $I^{\mathcal{A}}$ to be $\{2, 99\}$ and assume $\mathcal{S}$ ($\mathcal{R}$) is the answer computed by grounder for formula $E(x)$ (term $f(y)$, respectively). There will be two occurrences of MIN formula in the obtained ground formula (where γ is the empty assignment):

1. $MIN_{\mathcal{R},\mathcal{S},2}(\gamma)$.
2. $MIN_{\mathcal{R},\mathcal{S},99}(\gamma)$.

To translate $MIN_{\mathcal{R},\mathcal{S},99}(\gamma)$, the encoding described in the previous section translates $MIN_{\mathcal{R},\mathcal{S},2}(\gamma)$ too. Therefore there are two different sets of Tseitin variables both describing the same formula, $MAX(\mathcal{R}, \mathcal{S}, 2, \gamma)$. As discussed in Section 8.3, this kind of symmetry increases the solving time.

The author of this thesis proposed the following modifications to the EncodeAtomic method introduced in Algorithm 10. Instead of calling the EncodeMin method, as soon as we encounter a MIN atomic formula, Enfragmo saves the MIN formula along with a fresh propositional variable and returns the newly introduced propositional variable as the encoding of the MIN formula. When the MakeCNF phase has been finished, Enfragmo groups the MIN formulas with exactly the same answer to terms and formulas, together. Then, instead of encoding each MIN formula separately, Enfragmo computes transformations for MIN formulas in the same group.

The same technique has been used in Enfragmo for MAX, COUNT, SUM and the other place holders.

8.7 Experimental Evaluation

In this section, we compare the performance of FillAndReturn and standard Tseitin transformations. To do so, we implemented Tseitin transformation with all the optimization proposed in [33] in Enfragmo.

The main advantage of FillAndReturn transformation is that it merges the consecutive AND/OR operators. If the ground formula passed to the transformation does not have any consecutive AND/OR operator, FillAndReturn and Tseitin transformations generate very similar CNFs, if not exactly the same CNF.

The ground formula obtained by Enfragmo can have consecutive AND/OR connectives only if the specification has axioms in which:

1. There is a universal quantifier whose direct child is an AND node,
2. There is an existential quantifier whose direct child is an OR node,

3. There is an OR node (AND node) whose direct child is an OR node (AND node, respectively).

Some of the specifications presented in Chapter 3 have such axioms. We selected the following problems and compared the performance of the two transformations on different specifications for each of the problems: Blocked N-Queens, Hamiltonian Path, 15 Puzzle, Disjunctive Scheduling, Graph Colouring, Social Golfer and Hierarchical Clustering. As described in the previous chapters, for the Hamiltonian Path problem, computing fix-points for HP-03.T is time consuming, therefore we omitted it from our experiments.

Table 8.1 shows the performance FillAndReturn Transformation, FRT, and Tseitin Transformation, TT, on different specifications. There are two factors which should be considered when comparing the performance of transformations:

1. The running time of the transformation,
2. The running time of the SAT solver on the CNF obtained by the transformation.

Table 8.1 represents the average running time of the MakeCNF and Solving phases for each of the two transformations (without enabling the memorization option).

Specification	TT	FRT
BQ-01.T(118)	111/0.05/54.38	**112/0.058/97.63**
BQ-02.T(118)	118/0.00/15.31	**118/0.00/2.00**
BQ-03.T(118)	112/0.00/94.28	**114/0.00/95.11**
BQ-04.T(118)	118/0.21/17.84	**118/0.03/4.36**
HP-01.T(58)	58/1.45/48.79	**58/0.98/12.99**
HP-02.T(58)	58/0.70/28.98	**58/0.68/27.93**
HP-04.T(58)	**58/0.69/10.98**	58/0.69/11.32
15Puzzle.T(27)	**25/0.75/292.67**	27/0.73/380.38
DisjunctiveScheduling.T(10)	10/6.34/251.44	**10/5.70/87.54**
GC-01.T(106)	73/0.01/390.9	**73/0.00/360.96**
GC-02.T(106)	90/0.00/474.4	**90/0.00/434.4**
GC-03.T(106)	98/0.04/285.4	**100/0.03/278.7**
GC-04.T(106)	106/0.05/224.2	**106/0.04/211.0**
SG-01.T(175)	126/0.00/53.59	**129/0.00/69.44**
SG-02.T(175)	146/0.02/7.74	**152/0.01/17.03**
SG-03.T(175)	158/0.02/7.132	**163/0.03/12.58**
HierarichalClustering(12)	12/0.46/37.58	**12/0.38/10.86**

Table 8.1: The performance of Tseitin transformation vs FillAndReturn transformation (when the memorization option is disabled, and Enfragmo uses Static Variable Generator). The number in parenthesis in front of each specification is the number of instances. Each entry is in the form $n/t_1/t_2$, where n is the number of solved instances, and t_1 and t_2 are the average running time of the MakeCNF phase and Solving phase in seconds, respectively.

The general pattern in the data presented in Table 8.1 is that

- For all problems, the running time of the MakeCNF phase is much lower than the running time of Solving phase.
- Except for 15-Puzzle problem, FillAndReturn transformation performs better than Tseitin transformation, in the Solving phase. Either FillAndReturn transformation solves more instances or its average solving time is smaller than that of Tseitin transformation.
- FillAndReturn transformation never performs worse than Tseitin transformation in the MakeCNF phase.

We ran the same experiments with the memorization option enabled and found that enabling this option does not change the pattern.

8.8 Conclusion

In this chapter we described the general approach used by Enfragmo to convert a variable-free FO formula obtained from the grounding phase, to a CNF formula. We proposed some new modifications to the standard Tseitin transformation in Section 8.3 and also proposed a novel transformation, FillAndReturn transformation, for converting variable-free FO formulas to CNF in Section 8.4. We compared the performance of FillAndReturn and Tseitin transformations and showed that the proposed transformation performs better than Tseitin transformation, for almost all benchmarks we used in our experiments.

In this chapter we also explained how Enfragmo transforms an instance of MIN/MAX/BMIN/BMAX/BEQ/BLESSEQ place holders to CNF.

Chapter 9

Encoding For Cardinality Constraints

In this chapter, we show that a COUNT place holder, introduced in Section 6.5, can be translated as a cardinality constraint and then we propose two new encodings for translating cardinality constraints to CNF formulas.

9.1 Introduction

Given a set of variables $X = \{x_1, \cdots, x_n\}$, a non-negative integer k, and comparison operator $op \in \{<, \leq, =, \geq, >\}$, cardinality constraint $|\{x_1, \cdots, x_n\}|$ op k is a propositional constraint which restricts the number of *true* variables in set X. In this chapter, we focus on cardinality constraints whose comparison operator is "=". All the claims and techniques described in this chapter can be applied to the constraints whose operators are $<, \leq, >$ or $\geq$.

Enfragmo uses place holder $COUNT_{\mathcal{R},n}(\gamma)$, where $\mathcal{R}$ represents an answer to $\phi(\bar{x}, \bar{y})$, n is an integer, and γ is an assignment to variables in $\bar{y}$, to represent a formula equivalent to:

$$(Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\} = n)[\gamma]. \tag{9.1}$$

As in the previous chapters, we use $\mathcal{A}$ ($\mathcal{B}$) to denote the instance structure (expansion structure, respectively). In Chapter 6, we defined the $COUNT$ place holder such that structure $\mathcal{B}$ satisfies $COUNT_{\mathcal{R},n}[\gamma]$ iff exactly n formulas from the following set are *true* in structure $\mathcal{B}$:

$$\{\delta_{\mathcal{R}}(\gamma, \gamma') \mid \gamma' \text{ is an assignment to variables in } \bar{x}\}.$$

It should be clear that there is a close connection between an instantiation of the $COUNT$ place holder and a cardinality constraint. In Section 9.3, we demonstrate how to translate an instance of a $COUNT$ place holder to an instance of cardinality constraint, and in Section 9.5 we proposed two encodings for translating cardinality constraints to CNF. We also review some of existing encoding in Section 9.4.

As discussed in the previous chapter, different approaches for translation usually have different performances. The same is true for encodings of cardinality constraints. The following are some of the attributes which can be used to compare the performance of different encodings, theoretically:

- The number of variables generated by an encoding;

- The number of clauses generated by an encoding;
- Whether the generated CNF allows unit propagation to infer inconsistency efficiently.

Recall that the experiments presented in Chapter 3 show that Enfragmo performs well when the Count aggregate is used in the specifications. We believe this is, in part, because of the efficiency of the two proposed encodings for translating cardinality constraints to CNF. In this chapter, we show that unit propagation can infer facts from the CNF generated by our proposed encodings while it cannot infer the same facts on the CNF generated by the other encodings. Therefore, it can be expected that SAT solvers will perform better on the CNFs generated by our encodings. The result of our experiments Section 9.6 confirms that our proposed encodings perform better than the existing encodings.

9.1.1 My Contributions

The two new encodings, DP-based and DC-based encodings, introduced in Section 9.5, are novel and proposed by the author of this thesis [6]. Also the author showed, in Sections 9.5, that unit propagation infers more facts on the CNF generated by the two proposed encodings.

9.2 Notations and Definitions

In this section, we fix our notations and use them throughout the rest of the chapter. Also, we define when an encoding produces a *translation* for cardinality constraints.

9.2.1 Notations

In this chapter, we use the same notations used in Section 8.2, and here, we just introduce the notations which are specific to this chapter.

Let $X = \{x_1, \cdots, x_n\}$ be a set of Boolean variables. Cardinality constraint Q on X is specified as:

$$|\{x_1, \cdots, x_n\}| = k, \tag{9.2}$$

where $x_i \in X$, $0 \leq k \leq n$ is an integer, called *bound*.

Total assignment τ to X satisfies a cardinality constraint Q on X, $\tau \models Q$, iff τ maps exactly k variables out of $\{x_1, \cdots, x_n\}$ to *true*, i.e., $|\{x_i : \tau \models x_i\}| = k$.

Let $\phi(X)$ be a propositional formula, and τ be a assignment to variables in X. We use $\phi[\tau]$ to denote the formula obtained by replacing the variables in X with their corresponding values in τ.

9.2.2 Translation

A propositional constraint is a constraint involving propositional variables. A propositional constraint can be seen as a propositional formula which evaluates to *true* under the assignments that satisfy the constraint.

Definition 30 *A translation for propositional formula $\phi(X)$ is the pair $\langle v, C\rangle$, where v is a propositional variable, $C = \{C_1, \cdots, C_m\}$ is a set of clauses on $X \cup Y \cup \{v\}$ and Y is a set of (auxiliary) propositional variables where $v \notin Y$, such that*

- *For every total assignment τ to X, there is at least one assignment τ' such that $\tau \subseteq \tau'$ and $\tau' \models C$;*
- *Set of clauses C is such that $C \models \phi(X) \leftrightarrow v$.*

Intuitively, C describes the relation among input variables, $x \in X$, auxiliary variables, $y \in Y$, and the output variable, v. The set of clauses C is such that the truth value of v is the same as the truth value of $\phi(X)$ under all satisfying assignments for C.

Bailleux et al. defined a translation as a set of clauses [13]. It is easy to verify that these two definitions are equivalent. It is worth mentioning that our definition of a translation is not limited to cardinality constraints. We use the same definition in the next chapter too.

Example 30 *Let Q_1 be the cardinality constraint $|\{x_1, x_2, x_3\}| = 3$. Pair $\langle v_1, C_1 \rangle$ where C_1 is the following set of clauses is a* translation for Q_1:

$$\{\{\neg x_1, \neg x_2, \neg x_3, v_1\}, \{\neg v_1, x_1\}, , \{\neg v_1, x_2\}, \{\neg v_1, x_3\}\}.$$

Set of clauses C_1 is logically equivalent to $v_1 \leftrightarrow x_1 \wedge x_2 \wedge x_3$. In this translation, we introduced no auxiliary variable.

Example 31 *Let Q_2 be $|\{x_1, x_2, x_3\}| = 0$.*

- *The pair $\langle v_2, C_2 \rangle$ is a translation for Q_2, where C_2 is $\{\{x_1, x_2, x_3, v_2\}, \{\neg v_2, \neg x_1\}, \{\neg v_2, , \neg x_2\}, \{\neg v_2, \neg x_3\}\}$.*
- *The pair $\langle v_3, C_3 \rangle$ is another translation for Q_2, where C_3 is $\{\{x_1, x_2, y_1\}, \{\neg y_1, \neg x_1\}, \{\neg y_1, \neg x_2\}, \{\neg y_1, x_3, v_3\}, \{\neg v_3, \neg x_3\}, \{\neg v_3, y_1\}\}$.*

We used an auxiliary variable, y_1, in translation $\langle v_3, C_3 \rangle$ for Q_2.

Example 32 *Let Q_1 (Q_2) be the cardinality constraint of Example 30 (Example 31, respectively). To find assignment τ to X such that either τ satisfies Q_1 or τ falsifies Q_2, we can find a solution to a SAT instance having $C_1 \cup C_2 \cup \{v_1, \neg v_2\}$ as its set of clauses, where C_1, C_2, v_1 and v_2 are defined in the previous Examples.*

Note that using the notion of translation introduced in [13], there is no easy way to check whether there is such an assignment.

9.2.3 Unit Propagation

Unit propagation (UP) is a mechanism used by SAT solvers to accelerate the search process. UP applies a series of unit resolutions of assignment τ with set of clauses C. Unit resolution of set of clauses $C = C' \cup \{l_0, l_1, \cdots, l_k\}$ with assignment τ where $\tau \models \neg l_i$ for $i \geq 1$, extends τ such that $\tau \models l_0$. UP continuously applies unit resolution until the assignment τ does not change. This process can be formalized as follows:

Definition 31 *Let τ be an assignment, C be a set of clauses and l_i be some literals in C. We use $up(\tau, C)$ to denote the assignment obtained as the result of applying the unit resolution of τ with C.*

$$up(\tau, C) = \begin{cases} \tau \cup \{l\} & \exists l_0, l_1, \cdots, l_k, C', C = C' \cup \{l_0, \cdots, l_k\} \text{ s.t. } \tau \models \neg l_i, i \geq 1 \\ \tau & \text{otherwise.} \end{cases}$$

We denote the final assignment obtained as a result of unit propagation by $UP(\tau, C)$, which is equal to the fix point of $up(\tau, C)$.

In Definition 31, we used $\tau \cup \{l\}$ to represent the assignment which extends assignment τ and maps literal l to *true*.

9.3 Translating COUNT Place Holders to Cardinality Constraints

Recall that the place holder $COUNT_{\mathcal{R},n}(\gamma)$ where $\mathcal{R}$ represents an answer to $\phi(\bar{x}, \bar{y})$, n is an integer, and γ is an assignment to variables occurring in $\bar{y}$, is a place holder we used instead of a (large) formula which is equivalent to

$$(Count_{\bar{x}}\{\phi(\bar{x}, \bar{y})\} = n)\,[\gamma].$$

In this section, we show how Enfragmo generates an instance of cardinality constraint from an occurrence of COUNT place holder.

As in Chapter 6, we use assume the sort assigned to all variables is T. Having $\mathcal{R}$ and γ, let $S^{\gamma,n}_{\mathcal{R}}$ be the set of pairs $\langle \gamma_i, \phi_i \rangle$ where γ_i is an assignment mapping variables in $\bar{x}$ mapping to elements in T, and $\phi_i = \delta_{\mathcal{R}}(\gamma_i, \gamma)$. Our definition of an answer to a formula guarantees that ϕ_i is a formula equivalent to (and so equi-satisfiable with) $\phi[\gamma_i, \gamma]$.

In Chapter 8, we defined a transformation for a formula ϕ to be a pair $\langle v, C \rangle$ such that $C \cup \{v\}$ is satisfiable iff there is a structure satisfying ϕ. We use MakeCNF(ϕ) to denote the pair $\langle v, C \rangle$ produced by a transformation for ϕ.

We define set of pairs $C^{\gamma,n}_{\mathcal{R}}$ as

$$C^{\gamma,n}_{\mathcal{R}} = \{\langle v_i, C_i \rangle : \langle \gamma_i, \phi_i \rangle \in S^{\gamma,n}_{\mathcal{R}}, \text{ and } \langle v_i, C_i \rangle = \text{MakeCNF } (\phi_i)\}.$$

Let $VC^{\gamma,n}_{\mathcal{R}}$ and $CC^{\gamma,n}_{\mathcal{R}}$ be defined as

$$VC^{\gamma,n}_{\mathcal{R}} = \{v_i : \langle v_i, C_i \rangle \in C^{\gamma,n}_{\mathcal{R}}\}.$$
$$CC^{\gamma,n}_{\mathcal{R}} = \bigcup_{\langle v_i, C_i \rangle \in C^{\gamma,n}_{\mathcal{R}}} C_i.$$

Given assignment τ that satisfies all the set of clauses in $CC^{\gamma,n}_{\mathcal{R}}$, constraint $VC^{\gamma,n}_{\mathcal{R}}$ can be seen as a cardinality constraint which evaluates to *true* iff τ maps exactly n propositional variables in set $VC^{\gamma,n}_{\mathcal{R}}$ to *true*: Since the truth value of v_i corresponds to the truth value of $\phi[\gamma_i, \gamma]$, there exists an expansion structure, $\mathcal{B}$, that satisfies $COUNT_{\mathcal{R},n}(\gamma)$ iff there exists an assignment, τ, that satisfies all the clauses in $CC^{\gamma,n}_{\mathcal{R}}$ and also $\tau \models |VC^{\gamma,n}_{\mathcal{R}}| = n$.

9.4 Existing Encodings for Cardinality Constraints

In this section, we review two of the existing approaches for translating cardinality constraints of form Eqn (9.2) to SAT, and in the next section, we compare the performance of unit propagation on our proposed encodings and these two encodings.

9.4.1 BDD Encoding

A set of clauses which encodes a cardinality constraint can be constructed based on BDD representation of a cardinality constraint. Each BDD node is an if-then-else gate. The constraint $|\{x_1, \cdots, x_n\}| = k$ can be described using $\theta(n \times k)$ such nodes [33].

Here, we describe the BDD based encoding using the dynamic programming paradigm. Our description of BDD encoding and what has been described in [33] produce almost the same CNF. We selected this way of representing BDD based encoding since it allows us to compare this encoding with our proposed encodings.

In our construction, we use $(n+1) \times (k+1)$ fresh propositional variables, F_r^c, $r = 0 \cdots n$ and $c = 0 \cdots k+1$. We produce set of clauses C such that in all assignments τ which satisfies C, $\tau \models F_r^c$ iff $\tau \models |\{x_i \mid i \leq r\}| = c$. Set C contains the following unary clauses:

$$\begin{cases} \{F_r^c\} & \text{if } r \text{ and } c \text{ are both zero;} \\ \{\neg F_r^c\} & r = 0 \text{ and } c > 0. \end{cases}$$

Set C also contains the set of clauses generated by the standard Tseitin transformation to express the following formulas:

$$\begin{cases} F_{r+1}^{c+1} \leftrightarrow (F_r^c \wedge x_{r+1}) \vee (F_r^{c+1} \wedge \neg x_{r+1}). \\ F_{r+1}^0 \leftrightarrow F_r^0 \wedge \neg x_{r+1}. \end{cases}$$

The pair $\langle F_n^k, C \rangle$ is a translation for constraint $|\{x_1, \cdots, x_n\}| = k$.

9.4.2 Sorting-Network Encoding(SN)

The construction presented here is the adoption of the sorting-network encoding described in [33].

A sorting network is a circuit with n input wires and n output wires consisting of a set of comparators with two input wires and two output wires. A comparator compares its two inputs and outputs the greater one into its first output wire and the smaller one into its second output wire. Each output of a comparator is used as an input to another comparator except those used as output wires of the sorting network itself [16]. A comparator element can be defined as a circuit with two input wires, f_1 and f_2, and two output wires, o_1 and o_2, where $o_1 = f_1 \wedge f_2$ and $o_2 = f_1 \vee f_2$.

If the input to a sorting network contains Z zeros and O ones, the first O outputs of the sorting network are one and the rest of the wires are zero. Therefore, it can be concluded that there are exactly i ones in the input iff the value of the i-th output wire is one while the $i+1$-th output wire is zero.

Let S be a sorting network circuit, $x_1, \cdots, x_n$ be the input signal to S and $o_1, \cdots, o_n$ be the output signals of S. There are standard techniques to construct a SAT instance, in the form of CNF, from a Boolean circuit. The most commonly used approach to do so is to introduce a propositional variable for the output of each gate and, using some clauses, express the relation among the inputs and output of each gate. Let $\langle V_m, C_s \rangle$, for $i = 1 \cdots, n$, be a transformation describing the relation among the input signals $x_1, \cdots, x_m$ and the output signal o_m. Also let C be the set of clauses generated by Tseitin transformation to express the following propositional formula:

$$\begin{cases} V_m \leftrightarrow \neg o_1 & \text{m=0,} \\ V_m \leftrightarrow o_m \wedge \neg o_{m+1} & m < n, \\ V_m \leftrightarrow o_m & m = n. \end{cases}$$

The pair $\langle V_m, C_s \cup C \rangle$ is a translation for $|x_1, \cdots, x_n| = m$.

9.5 Proposed Encodings

In this section, we propose two new encodings for translating cardinality constraints, and hence $COUNT$ place holders, to SAT.

9.5.1 Dynamic-Programming Based Encoding (DP)

In this encoding, we construct the necessary and sufficient conditions for the satisfiability of $|\{x_1, \cdots, x_i\}| = c$, by using the conditions for satisfiability of $|\{x_1, \cdots, x_{i-1}\}| = c$ and $|\{x_1, \cdots, x_{i-1}\}| = c - 1$.

Let D_i^c, $0 \leq c \leq i \leq n$, be fresh Boolean variables. We generate set of clauses C such that for any assignment τ satisfying C, we have $\tau \models D_i^c$ iff $\tau \models |\{x_1, \cdots, x_i\}| = c$, for $0 \leq c \leq i \leq n$. To do so, we use the following construction:

1. If both x_i and D_{i-1}^c are *true*, D_i^{c+1} must be *true*: can be expressed using clause $\{\neg x_i, \neg D_{i-1}^c, D_i^{c+1}\}$, which asserts $\left(x_i \wedge D_{i-1}^c\right) \rightarrow D_i^{c+1}$;
2. If x_i is *false*, and D_{i-1}^c is *true*, D_i^c must be *true*: can be expressed using clause $\{x_i, \neg D_{i-1}^c, D_i^{c+1}\}$, which asserts $\left(\neg x_i \wedge D_{i-1}^c\right) \rightarrow D_i^{c+1}$;
3. If x_i is *true*, and D_i^{c+1} is *false*, D_{i-1}^c must be *false*: can be expressed using clause $\{\neg x_i, D_i^{c+1}, \neg D_{i-1}^c\}$, which asserts $\left(x_i \wedge \neg D_i^{c+1}\right) \rightarrow \neg D_{i-1}^c$;
4. If x_i is *false*, and D_i^c is *false*, D_{i-1}^c must be *false*: can be expressed using clause $\{x_i, D_i^c, \neg D_{i-1}^c\}$, which asserts $(\neg x_i \wedge \neg D_i^c) \rightarrow \neg D_{i-1}^c$;
5. Exactly one of D_i^c, $0 \leq c \leq i$, can be *true*: can be expressed using clause $\{\neg D_i^{c_1}, \neg D_i^{c_2}\}$, $c_1 \neq c_2$, which asserts $(D_i^{c_1} \rightarrow \neg D_i^{c_2})$;
6. If both D_{i-1}^{c-1} and D_{i-1}^c are *false*, D_i^c must be *false*: can be expressed using clause $\{D_{i-1}^{c-1}, D_{i-1}^c, \neg D_i^c\}$, which asserts $\left(\neg D_{i-1}^c \wedge \neg D_{i-1}^{c-1}\right) \rightarrow \neg D_i^c$;
7. If D_i^{c-1} and D_i^c are *true*, then x_i must be *true*. This can be expressed using clause $\{\neg D_{i-1}^{c-1}, \neg D_i^c, x_i\}$, which asserts $\left(D_{i-1}^{c-1} \wedge D_i^c\right) \rightarrow x_i$;

8. If D^c_{i-1} and D^c_i are *true*, then x_i must be *false*. This can be expressed using $\{\neg D^c_{i-1}, \neg D^c_i, \neg x_i\}$, which asserts $\left(D^c_{i-1} \wedge D^c_i\right) \rightarrow \neg x_i$.

We must also add unary clause $(\{D^0_0\})$ to assert

$$|\{\}| = 0.$$

In the above encoding, the two sets of clauses (7) and (8), described above, are not necessary and can be omitted. Having these extra clauses allows unit propagation to infer extra facts which are not inferable without these clauses.

Let C be the set of clauses generated to assert the relationship among D^c_i and x_i variables. Then the pair $\langle D^k_n, C\rangle$ is a translation for $|\{x_1, \cdots, x_n\}| = k$.

The idea used in developing this encoding is very similar to that of BDD encoding. The main difference between these two encodings is that DP encoding does not need to introduce any new Tseitin variables, while BDD encoding must introduce $2nk$ extra Tseitin variables, one for $(F^c_r \wedge x_{r+1})$ and one for $(F^{c+1}_r \wedge \neg x_{r+1})$, $r = 1 \cdots n, c = 1 \cdots k$.

We know that both F^c_i, in BDD encoding, and D^c_i, in DP encoding, assert that exactly c out of the first i input variables are mapped to *true*. Since the set of variables in BDD encoding and DP encoding are not exactly the same, there is no standard way to compare the performance of unit propagation, theoretically. Restricting our attention to the variables in the two encodings, which assert the same facts, i.e., F^c_i and D^c_i enables us to compare the performance of unit propagation on these two encodings.

The following proposition shows that although both BDD and DP encodings implement a similar idea, there are facts unit propagation can infer on the CNF generated by DP encoding but not on the CNF generated by BDD encoding.

Proposition 22 *Let X be set of propositional variables $\{x_1, \cdots, x_n\}$, Q be a cardinality on X, $|X| = k$, and $\langle v_1, C_1\rangle$ ($\langle v_2, C_2\rangle$) be the translation obtained from BDD encoding (DP encoding, respectively). Let τ_1 and τ_2 be (partial) assignments to propositional variables in X. We assume τ_1 and τ_2 assert the same set of facts, i.e., $\tau_1 \models x_i$ iff $\tau_2 \models x_i$, and $\tau_1 \models \neg x_i$ iff $\tau_2 \models \neg x_i$.*

If UP infers the truth value of F^j_i when it runs on C_1 with assignment τ_1, UP infers the same truth value for D^j_i, when it runs on C_2 with assignment τ_2. In addition, if UP infers the truth value of x_i when it runs on C_1 with assignment τ_1, UP infers the same value for x_i when it runs on C_2 with assignment τ_2.

Proof: In this proof, we use $\tau_2 \supseteq \tau_1$ to assert

1. If τ_1 maps F^c_i to either *true* or *false*, τ_2 maps D^c_i to the same value.
2. If τ_1 maps x_i to either *true* or *false*, τ_2 maps x_i to the same value.

To prove this proposition, we show that given constrain Q, translations $\langle v_1, C_1\rangle$ and $\langle v_2, C_2\rangle$ and two assignments τ_1 and τ_2, we have $UP(\tau_2, C_2) \supseteq UP(\tau_1, C_1)$.

Let $Rank_1(\tau_1)$ denote the maximum i such that τ_1 assigns a value to x_i. If there is no such i, we define $Rank_1(\tau_1)$ to be 0. We define $Rank_2(\tau_2)$ similarly. Since both τ_1 and τ_2 are mapping elements in X to *true/false* and they agree on their assignments, we have $Rank_1(\tau_1) = Rank_2(\tau_2)$. We use strong induction on $Rank_1(\tau_1)$ to prove this proposition:

The base case is when $Rank_1(\tau_1)$ is zero. This means τ_1 is an empty assignment and since there is no unary clause in C_1, we have $UP(\tau_1, C_1) = \tau_1$. Using the same argument, we have $UP(\tau_2, C_2) = \tau_2$. And so, we have $UP(\tau_2, C_2) \supseteq UP(\tau_1, C_1)$.

Let us assume the proposition holds for every pair of assignments, τ_1 and τ_2 satisfying the requirement on the assignments where $Rank_1(\tau_1) \leq l$. We show that the proposition holds when assignments τ_1 and τ_2 have $Rank_1(\tau_1) = Rank_2(\tau_2) = l + 1$.

Let τ_1' (τ_2') be the assignment which is the same as τ_1 (τ_2, respectively) but does not map x_{l+1} to any value. So, $Rank_1(\tau_1') = Rank_2(\tau_2') \leq l$. By induction hypothesis, we have $UP(\tau_2', C_2) \supseteq UP(\tau_1', C_1)$. Also, from the properties of unit propagation, we have $UP(\tau_1', C_1') \subseteq UP(\tau_1, C_1)$ and $UP(\tau_2', C_2') \subseteq UP(\tau_2, C_2)$.

Let's assume $\tau_1 \models x_{l+1}$ and there are i and c such that $UP(\tau_1', C_1)$ does not assign any value to F_i^c but $UP(\tau, C_1)$ does. We show $UP(\tau_2, C_2)$ assigns the same value to D_i^c.

Since we assumed τ_1 is an assignment on X and $\tau_1 \models x_{l+1}$, it is straightforward to verify that the assignments $UP(\tau_1, C_1)$ and $UP(\tau_1', C_1)$ agree on the values they assign to $F_{i'}^{c'}$, $c' \leq i' \leq l$. Also $UP(\tau_2', C_2) \supseteq UP(\tau_1', C_1)$ implies that for every $c \leq i \leq l$ if $UP(\tau_1, C_1) \models F_i^c$ then $UP(\tau_2, C_2) \models D_i^c$.

Therefore, we only need to show that if there is F_{l+1}^c which is mapped to *true* by $UP(\tau_1, C_1)$, then $UP(\tau_2, C_2)$ also maps D_{l+1}^c to *true*.

From the construction presented in Section 9.4.1 if $UP(\tau_1, C_1) \models F_{l+1}^c$ and $\tau_1 \not\models F_{l+1}^c$, having $UP(\tau_1, C_1) \models x_{l+1}$, we conclude that $UP(\tau_1, C_1) \models F_l^{c-1}$.

Now, we show $UP(\tau_2, C_2) \models x_{l+1}$ and $UP(\tau_2, C_2) \models D_l^{c-1}$:

1. We assumed $\tau_1 \models x_{l+1}$ and so, we have $\tau_2 \models x_{l+1}$, which implies $UP(\tau_2, C_2) \models x_{l+1}$.
2. By induction hypothesis, we have if $UP(\tau_1, C_1) \models F_l^c$, then $UP(\tau_2, C_2) \models D_l^c$.

Then from set of clauses $\{\neg x_{l+1}, \neg D_l^{c-1}, \neg D_{l+1}^c\}$, UP infers that D_{l+1}^c must be *true*.

There are three other cases which need to be considered to complete the proof:

- $\tau_1 \models x_{l+1}$ and $UP(\tau_1, C_1) \models \neg F_i^c$;
- $\tau_1 \models \neg x_{l+1}$ and $UP(\tau_1, C_1) \models F_i^c$;
- $\tau_1 \models \neg x_{l+1}$ and $UP(\tau_1, C_1) \models \neg F_i^c$.

The proofs for these three cases are similar to what has been presented here. ■

In Proposition 22, we assumed that τ_1 is an assignment to variables in X. It can be shown that $UP(\tau_2, C_2) \supseteq UP(\tau_1, C_1)$ if we allow τ_1 to be an assignment to $X \cup \{F_i^c\}$ and τ_2 to be an assignment to $X \cup \{D_i^c\}$, such that τ_1 and τ_2 assert the same set of facts.

So far we have shown that the set of facts unit propagation can infer on the translation obtained by DP encoding is a superset or equal to the set of facts inferable from the translation obtained by BDD encoding. The following proposition shows that there are facts inferable by unit propagation, on the CNF generated by DP encoding, which are not inferable by unit propagation, on the CNF generated by BDD encoding.

Proposition 23 *There are facts which unit propagation can infer on the CNF generated by DP encoding but not on the CNF generated by BDD encoding.*

Proof: Let Q be $|\{x_1, x_2, x_3\}| = 2$, $\tau_1 = \{\neg F_2^0, \neg F_2^1\}$ and $\tau_2\{\neg D_2^0, \neg D_2^1\}$. Let $\langle v_1, C_1\rangle$ ($\langle v_2, C_2\rangle$) be translation for Q using BDD encoding (DP encoding, respectively).

By applying unit propagation to C_1 with τ_1, we have $UP(\tau_1, C_1) = \tau_1$ (UP cannot infer any new fact).

By applying unit propagation to C_2 with τ_2, we have $UP(\tau_2, C_2) = \tau_2 \cup \{D_2^2, D_1^1, x_2, x_1, \neg D_1^0\}$.

So, we have $D_2^2 \in UP(\tau_2, C_2)$ while $F_2^2 \notin UP(\tau_1, C_1)$. This means that, on the CNF generated by DP encoding, unit propagation infers $|\{x_1, x_2\}| = 2$ (also x_1 and x_2 must be *true*) if we have $|\{x_1, x_2\}| \neq 0$ and $|\{x_1, x_2\}| \neq 1$. On the other hand, UP can infer no fact on the CNF generated by BDD encoding.

■

9.5.2 Divide-and-Conquer Based Encoding (DC)

Given cardinality constraint $|\{x_1, \cdots, x_m\}| = n$, the idea of this encoding is to divide the variables in set $X = \{x_1, \cdots, x_m\}$ into two sets $X_1 = \{x_1, \cdots, x_{\frac{m}{2}}\}$ and $X_2 = \{x_{\frac{m}{2}+1}, \cdots, x_m\}$, and then find the conditions describing how many variables from each subset is True. In the next step, appropriate conditions would be merged, creating the condition for the set X. More formally, let $D^c_{(s,e)}$ be the Tseitin variable which is *true* in assignment τ iff $|\{x_i : s \leq i \leq e \wedge \tau \models x_i\}| = c$. So, $D^c_{(1,n)}$ is the necessary and sufficient condition for having exactly c variables evaluated as *true* out of set X, e.g., $|\{x_1, \cdots, x_m\}| = c$. Then the relation among variables $D^c_{(s,e)}$ is defined using sets of clauses equivalent to the following propositional formulas:

1. If $D^{c_1}_{(s,m)}$ and $D^{c_2}_{m+1,e}$ are *true*, $D^{c_1+c_2}_{(s,e)}$ must be *true*: $\left(D^{c_1}_{(s,m)} \wedge D^{c_2}_{(m+1,e)}\right) \rightarrow D^{c_1+c_2}_{(s,e)}$;
2. At most one of $D^j_{(s,e)}$, $j = 0, \cdots, m$, can be *true*: $D^{c_1}_{(s,m)} \rightarrow \neg D^{c_2}_{(s,m)}$;
3. If $D^c_{(s,e)}$ is *false*, and $D^{c_1}_{(s,m)}$ is *true*, $D^{c-c_1}_{(m+1,e)}$ must be *false*: $(\neg D^c_{(s,e)} \wedge D^{c_1}_{(s,m)}) \rightarrow \neg D^{c-c_1}_{(m+1,e)}$;
4. If $D^c_{(s,e)}$ is *false*, and $D^{c_2}_{(m+1,e)}$ is *true*, $D^{c-c_2}_{(s,m)}$ must be *false*: $\left(\neg D^c_{(s,e)} \wedge D^{c_2}_{(m+1,e)}\right) \rightarrow \neg D^{c-c_2}_{(s,m)}$.

Also, we need to add unary clause $(\{D^0_{(0,0)}\})$, stating:

$$|\{\}| = 0.$$

And also we need to add some binary clauses stating:

- $D^0_{(s,s)} = \neg x_s$;
- $D^1_{(s,s)} = x_s$.

In the above encoding, the two sets of clauses (3) and (4) are not necessary and can be omitted. Having those extra clauses increases the facts that unit propagation can infer.

Let C be the above set of clauses, then pair $\langle D^n_{1,m}, C\rangle$ is a translation for $|\{x_1, \cdots, x_m\}| = n$.

Proposition 24 *Unit propagation performs better on the CNF generated by DC encoding comparing to the CNF generated by sorting network encoding.*

The proof idea of the proof is very similar to that of Proposition 23. We construct an assignment which asserts the same facts for both encodings and show that unit propagation can infer facts on the CNF produced by DC encoding while it cannot infer the same facts on the CNF produced by the sorting network.

9.6 Experimental Evaluation

In this section, we compare the performance of our proposed encodings for cardinality constraints against Sorting Networks (SN) encoding, BDD encoding. Also, we use MXC, a SAT+Cardinality solver, as the ground solver engine for Enfragmo.

As we described in the previous sections, DP and BDD encodings are almost the same, except that DP encoding writes some extra clauses in order to enable unit propagation to infer inconsistency earlier, and also it introduces fewer propositional variables. Among the specifications presented in Chapter 3, we used Count aggregate in the Blocked N-Queens and Social Golfers problems. In Chapter 3, we used DC encoding to translate the cardinality constraints. In this section, we compare the performance of different encodings on the Blocked N-Queens and Social Golfers problems. We have selected specifications SG-01, SG-02 and SG-03 for the Social Golfers problem and BQ-3 and BQ-4 for the Blocked N-Queens problem, and run Enfragmo using five different approaches to handle the cardinality constraints.

1. BDD: the cardinality constraints are translated into CNF using BDD encoding, Subsection 9.4.1.
2. DP: the cardinality constraints are translated into CNF using DP encoding, Subsection 9.5.1.
3. DC: the cardinality constraints are translated into CNF using DC encoding, Subsection 9.5.2.
4. SN: the cardinality constraints are translated into CNF using SN encoding, Subsection 9.4.2.
5. CARD: the cardinality constraints are written directly into CNF, and then Enfragmo uses MXC, which is a SAT+Cardinality solver, to solve the CNF.

Table 9.1 presents the performance of these five configurations.

Spec	BDD	DP	DC	SN	CARD
SG-01(175)	125/61.0	125/60.6	130/72.8	128/85.7	101/39.9
SG-02(175)	151/42.5	151/41.2	150/19.7	150/21.6	125/54.1
SG-03(175)	163/29.6	163/29.3	162/7.0	162/13.3	139/48.9
BQ-02(400)	300/10.9	300/15.2	300/27.4	300/12.7	300/89.5
BQ-04(400)	397/34.3	397/27.4	400/41.2	397/27.88	389/136.6

Table 9.1: Performance of different methods for handling cardinality constraints. The number in parenthesis is the number of instances. Each entry is in the form n/t where n is the number of solved instances, t is Solving phase running times in second

From the data presented in Table 9.1, we can observe the following:

1. With regard to the Solving times, DP encoding performs better than BDD encoding in almost all of our experiments: This observation may be due to the extra clauses in the DP encoding that enables unit propagation to infer inconsistencies earlier.
2. DC encoding performs better than SN encodings, in almost all of our experiments: This may be due to the depth of the corresponding circuits. The circuit generated by the sorting network is deeper than the one generated by DC.
3. In all of the experiments, MXC performed the worst.
4. Overall, there is no clear winner between DP and DC encodings: DC generates a shallower circuit, but on the other hand, it produces larger clauses, while DP generates a relatively deep circuit but it produces small clauses (binary and ternary clauses).

In all the experiments presented in this chapter, we used MXC as the ground solver. In another set of experiments, we used MiniSAT to compare the performance of BDD, DP, DC and SN encodings. It was interesting to observe that MXC performs on average three times faster than MiniSAT.

Chapter 10

Encoding For Pseudo-Boolean Constraints

A Pseudo-Boolean constraint (PB-constraint) is a linear constraint over Boolean literals. PB constraints are widely used to express NP-complete problems. The SUM place holders, introduced in Chapter 8, can also be easily transformed to PB-constraint.

This chapter introduces a family of algorithms for translating Pseudo-Boolean constraints into CNF formulas. These algorithms are centered around the idea of rewriting a PB-constraint as the conjunction of a set of easier to translate constraints, we call them PBMod-constraints. The CNF produced by the proposed encoding has small size, and unit propagation can derive facts that it cannot derive using the CNF formulas obtained by other commonly-used transformations.

We present the results of an experimental performance comparison of our method with other methods, using instances of the number partitioning problem. The results show that our method is superior in certain parameter ranges.

10.1 Introduction

A Pseudo-Boolean constraint (PB-constraint), which is also known as a 0-1 integer linear constraint by the integer linear programming community, is a generalization of a clause. A PB-constraint is an inequality or equality on a linear combination of Boolean literals:

$$\sum_{i=1}^{n} a_i l_i \{<, \leq, =, \geq, >\} b,$$

where $a_1, \cdots a_n$ and b are integers and $l_1, \cdots, l_n$ are literals. The left-hand side of a PB-constraint under truth assignment τ is equal to the sum of the coefficients whose corresponding literals are mapped to *true* by τ.

One way to build a solver for sets of PB-constraints is to develop a solver capable of handling PB-constraints natively. PBS [10] and PUEBLO [61] are examples of such solvers. Another approach is to rewrite a given PB-constraint with a logically equivalent set of clauses and then use a SAT solver to find a solution. The main benefit of the the second approach is that every SAT solver, even those which are going to be developed in the future, can be plugged in to the system.

In this chapter, we propose a method which works based on the latter approach, i.e., rewriting PB-constraints as a set of clauses. Our main motivation for choosing this direction is that we do not want Enfragmo to be dependent on a specific SAT solver but want it to able to use any SAT solver as its low-level solver engine.

There are certain NP problems which can be expressed as the combination of a relatively small CNF formula plus one or two PB-constraints. The Vehicle Routing Problem and its variations [48] and the Knapsack problem are examples of such problems. Having a *good* translation for PB-constraints allows users to attack these problems using SAT solvers. Most professional users encode these problems using Integer Linear Programming (ILP) tools. Although it is well-known that every NP problem can be expressed as an ILP program, unfortunately, there is no *natural* way to express certain sentences as a set of linear equality/inequalities, for example, properties involving disjunctions of constraints, such as "Either John or Maria is wearing a green shirt and a black hat". The standard techniques for expressing these properties involves introducing additional variables, but extensive use of these techniques causes performance problems. As explained in Chapter 8, Tseitin transformation also introduces additional variables to translate these kinds of properties to propositional CNFs but the performance penalty seems to be little.

We define a PBMod-constraint to be:

$$\sum_{i=1}^{n} a_i l_i \equiv b \ (\text{mod M}),$$

where $a_1, \cdots a_n$ and b are non-negative integers less than M, and $l_1, \cdots, l_n$ are literals.

We show that a given PB-constraint can be rewritten as the conjunction of a set of appropriately selected PBMod-constraints. So, in order to translate PB-constraints to a CNF formula, we need to determine how to *choose* the set of PBMod-constraints and how to *translate* a PBMod-constraint to a CNF formula. As we show in this chapter, there are many PB-constraints whose unsatisfiability can be proven by showing the unsatisfiabiliy of a PBMod-constraint. This chapter presents two methods for translating PBMod-constraints to CNF. Both these encodings allow unit propagation to infer inconsistency if the current assignment cannot be extended to a satisfying assignment for that PBMod-constraint and hence unit propagation can infer inconsistency for the original PB-constraint. We also show that the number of PB-constraints for which unit propagation can infer inconsistency, given the CNF formula generated by our proposed encoding, is much larger than the other existing encodings. Also, we prove that it is impossible to translate all PB-constraints in the form $\sum a_i l_i = b$ into polynomial size arc consistent CNFs unless P=NP.

We use the number partitioning problem as the benchmark to compare the performance of our proposed translations with the other commonly used translations in the SAT community. The results of the experiments confirm that our translations performs better than the other methods.

The structure of this chapter is as follows: The next section is devoted to background and definitions. The proposed encodings are presented in sections 10.3 and 10.4. In Section 10.5, four existing translations (encodings) for converting a PB-constraint to CNF are described. In Section 10.6, we study the performance of unit propagation on the resulting CNF of proposed encodings. Specifically, we identify a class of PB-constraints for which our proposed encodings are arc consistent. We also show that there is no polynomial size arc consistent encoding for PB-constraint in the form $\sum a_i l_i = b$ unless P=NP. Section 10.7 compares the performance of different encodings on the instances of the subset sum problem.

10.1.1 My Contributions

The idea of translating a PB-constraint to a set of PBMod-constraint is contributed by the author. Finding the set of the modulos, Section 10.3, and all the encodings for PBMod-constraints, Section 10.4, the proof for co-NP-hardness and the upper/lower bounds presented in Section 10.6, are author's contributions [3].

10.2 Background

In this section, we fix our notations and use them through the rest of the chapter.

10.2.1 Notations

We use the same notation for assignments, clauses and etc, as were introduced in Section 8.2.

Let X be a set of Boolean variables. PB-constraint Q on X is specified as:

$$a_1 l_1 + \cdots + a_n l_n \qquad \{<, \leq, =, \geq, >\} \qquad b,$$

where a_i is the integer *coefficient* of l_i, b is an integer, called the *bound*, and l_i is a literal, such that, var(l_i)$\in X$.

Total assignment τ to X satisfies a PB-constraint Q on X, written as $\tau \models Q$, if the value of left-hand side of Q under τ, i.e., $\sum_{i:\tau \models l_i} a_i$, and that of the right-hand side of Q satisfy the comparison operator.

10.2.2 Translation

Note that a PB-constraint on X can be seen as a formula which evaluates to *true* on assignments that satisfy the constraint, and *false* otherwise. Therefore we can use Definition 30 to define a translation for PB-constraints (To make this chapter self-contained, we repeat that Definition).

Definition 32 *Given constraint $\phi(X)$, where $X = \{x_1, \cdots, x_n\}$ is a set variables, we call the pair $\langle v, C\rangle$, where v is a Boolean variable, $C = \{C_1, \cdots, C_m\}$ is a set of clauses on $X \cup Y \cup \{v\}$ and Y is a set of (auxiliary) propositional variables, disjoint from X, such that $v \notin Y$, such that*

- *For every total assignment τ to X, there is at least one assignment τ' such that $\tau \subseteq \tau'$ and $\tau' \models C$;*
- *Set of clauses C is such that $C \models \phi(X) \leftrightarrow v$.*

Example 33 *Let Q be the following unsatisfiable PB-constraint. The pair $\langle v, \{C_1\}\rangle$ where $C_1 = \{\neg v\}$ is a* translation for Q.

$$2x_1 + 4\neg x_2 = 3.$$

As in the previous chapter, if the set of clauses is clear from the context, we represent a translation using just v, instead of $\langle v, C\rangle$.

10.2.3 Canonical Form

Let us consider the following PB-constraint:

$$a_1 l_1 + \cdots + a_n l_n = b, \tag{10.1}$$

where b and all coefficients are positive. We show that every PB-constraint can be rewritten as a PB-constraint in form of 10.1.

Definition 33 *Constraints Q_1 on X and Q_2 on $Y \supseteq X$ are equivalent iff, for every satisfying assignment τ for Q_1, there exists at least one expansion of τ to Y satisfying Q_2, and for every total assignment τ to X which does not satisfy Q_1, every expansion of τ to Y falsifies Q_2.*

Observation 5 *Let $n \geq 1$. The following PB-constraints are equivalent.*

1. $\sum_{i=1}^{n} a_i l_i >= b$,
2. $\sum_{i=1}^{n} a_i l_i > b - 1$,
3. $\sum_{i=1}^{n} -a_i l_i <= -b$,
4. $\sum_{i=1}^{n} -a_i l_i < 1 - b$.

Observation 6 *Let m and n be such that $1 \leq m \leq n$. Then (1) and (2) are equivalent.*

1. $\sum_{i=1}^{m-1} a_i l_i + a_m l_m + \sum_{i=m+1}^{n} a_i l_i < b$,
2. $\sum_{i=1}^{m-1} a_i l_i - a_m \neg l_m + \sum_{i=m+1}^{n} a_i l_i < b - a_m$.

Observation 5 and Observation 6 imply that every PB-constraint whose comparison operator is in $\{\leq, <, >, \geq\}$ can be rewritten as an equivalent PB-constraint with positive coefficients in the following form:

$$\sum_{i=1}^{n} a_i l_i < b. \tag{10.2}$$

If the right-hand side of PB-constraint 10.2, b, is less than or equal to zero, no assignment satisfies the constraint, i.e., the pair $\langle v, \{\{\neg v\}\}\rangle$ can be used as a translation for this constraint. If we have a PB-constraint of from (10.2) with $b = 1$, pair $\langle v, C\rangle$, where C is any set of clauses equivalent to $v \leftrightarrow \neg l_1 \wedge \cdots \wedge \neg l_n$, is a translation for that constraint.

Proposition 25 *Let $n \geq 1$, each $a_i \geq 0$, $b > 1$ and $B = \lfloor \log_2 b \rfloor$. Also, assume the variables y_i are fresh Boolean variables. The following PB-constraints are equivalent.*

1. $\sum_{i=1}^{n} a_i l_i < b$,
2. $\sum_{i=1}^{n} a_i l_i + \sum_{i=0}^{B} 2^i y_i = b - 1$.

Proof: Let Q_1 denote $\sum_{i=1}^{n} a_i l_i < b$, and Q_2 denote $\sum_{i=1}^{n} a_i l_i + \sum_{i=0}^{B} 2^i y_i = b - 1$. Assume Q_1 is a PB-constraint on X.

1. Let τ be a total assignment to X satisfying Q_1, and c denote the value of $\sum_{i:\tau \models l_i} a_i$. Therefore $c \leq b - 1$. We construct assignment τ' such that $\tau' \models Q_2$:

   $\tau' \models y_i$ iff the i-th bit of $(b-1) - c$, in the binary representation, is one.

2. Let total assignment τ falsify Q_1, and b' be the value of $\sum_{i:\tau \models l_i} a_i$. Since τ falsifies Q_1, we have $b' \geq b$. Let τ' be a total assignment to $X \cup Y$ which expands τ. The value of the left-hand side of Q_2, under τ', is greater than or equal to $\sum_{i:\tau' \models l_i} a_i = \sum i : \tau \models l_i a_i = b'$. So, no matter what τ' assigns to variables in Y, the left-hand side of Q_2 evaluates to an integer greater than or equal to b, and hence, τ' does not satisfy Q_2.

We showed that given a satisfying assignment for Q_1, there is an expansion of that assignment which satisfies Q_2. We also proved that given an assignment which does not satisfy Q_1, any of its expansions falsifies Q_2. So, by Definition 33, Q_1 and Q_2 are equivalent.

■

In conclusion, every PB-constraint can be rewritten as an equivalent PB-constraint in the form of Equation 10.1. So, if we are able to translate the PB-constraints in this form, we can translate every PB-constraint as well.

10.2.4 Unit Propagation

In this chapter, we follow the notation introduced in Subsections 8.2.1 and 9.2.3.

Generalized arc consistency (GAC) is one of the desired properties for an encoding which is related to the performance of unit propagation. Bailluex et al., in [13], defined UP-detect inconsistency and UP-maintain GAC for PB-constraint's encodings. Although, that definition of a *translation* is slightly different from ours, these two concepts can still be discussed in our context.

Let E be an encoding for PB-constraints, Q be a PB-constraint on X and $\langle v, C \rangle$ the translation for Q obtained from encoding E. Then,

1. *UP-detect inconsistency:* Encoding E for constraint Q supports UP-detect inconsistency if for every (partial) assignment τ, such that $\tau[X] \not\models Q, UP(\tau, C) \models \neg v$.
2. *UP-maintain GAC:* Encoding E for constraint Q supports UP-maintain GAC if for every (partial) assignment τ and variable $x \in X$, such that x has the same value in all assignments extending τ and satisfying Q, $UP(\tau, C \cup \{v\}) \models x$ (or $UP(\tau, C \cup \{v\}) \models \neg x$, depending on the truth value of x).

An encoding for PB-constraints is generalized arc consistent, or simply, arc consistent, if it supports both UP-detect inconsistency and UP-maintain GAC for all possible constraints.

The procedure of converting a SUM place holder to a PB-constraint is similar to what is presented in Section 8.5.1 and we do not describe the procedure here.

10.3 Proposed Method

In this section, we explain how our proposed approach works for PB-constraints in form (10.1).

Let a normal PBMod-constraint be an expression of the following form:

$$\sum_{i=1}^{n} a_i l_i \equiv b \pmod{\mathrm{M}}, \tag{10.3}$$

where $0 \leq a_i < M$ for all $1 \leq i \leq n$ and $0 \leq b < M$. Total assignment τ is a solution to a PBMod-constraint iff the value of left-hand side summation under τ minus the value of the right-hand side of the equation, b, is a multiple of M.

Definition 34 *Let Q be the PB-constraint $\sum a_i l_i = b$ and M an integer greater than 1. We use $Q[M]$ to denote the PBMod-constraint $\sum a'_i l_i \equiv b' (mod\ M)$ where:*

1. $a'_i = a_i$ *mod* M,
2. $b' = b$ *mod* M.

Example 34 *Let Q be $6x_1 + 5x_2 + 7x_3 = 12$. By Definition 34, we have:*

1. $Q[3]$ *is* $0x_1 + 2x_2 + 1x_3 \equiv 0 (mod\ 3)$.
2. $Q[5]$ *is* $1x_1 + 0x_2 + 2x_3 = 2 (mod\ 5)$.

Every satisfying assignment for Q also satisfies each $Q[M]$, for $M \geq 2$. Also, for large enough value of M, when M is greater than $\sum a_i$, every solution for $Q[M]$ is also a solution for Q. So, for the appropriate values of M, two constraints Q and $Q[M]$ have the same set of solutions. We propose two ways to select the value of M such that translating the corresponding PBMod-constraint is easier than translating the original PB-constraint.

Lemma 2 *For any PB-constraint Q in form $\sum a_i l_i = b$, if $M > \sum a_i$, PBMod-constraint $Q[M]$ and PB-constraint Q have exactly the same set of satisfying assignments, i.e., any assignment either satisfies both constraints or neither of them.*

Proof: If τ is a solution (a satisfying assignment) for Q, τ satisfies $Q[M]$, too. Now, let τ satisfy $Q[M]$. The value of the left-hand side of $Q[M]$ under τ is an integer in the form $b + k * M$ for some $k \geq 0$. As we have $0 \leq b + k * M \leq \sum a_i < M$, we can infer that k must be zero and so the sum of left-hand side of $Q[M]$ under τ is exactly equal to b. ■

Lemma 3 *Let Q denote PB-constraint $\sum a_i l_i = b$. Also, let M_1 and M_2 be two integers and $M_3 = lcm\ (M_1, M_2)$. Assume S_j is the set of assignments satisfying $Q[M]$ when $M = M_j$, for $j = 1, 2$ and 3. Therefore:*

$$S_3 = S_1 \cap S_2.$$

Proof: Let $\tau \in S_3$. By definition, τ satisfies both $Q[M_1$ and $Q[M_2]$. Let Sum denote the value of $\sum a_i l_i$, under assignment τ. Since $\tau \models Q[M_1]$ ($\tau \models Q[M_2]$), $Sum \equiv b$ (mod M_1) ($Sum \equiv b$ (mod M_2)).

Since, $Sum \equiv b$ (mod M_1) and $Sum \equiv b$ (mod M_2), $Sum \equiv b$ (mod M3). So, the left-hand side of Q evaluates to b, modulo M_3, under assignment τ. ■

Lemma 3 tells us that in order to find the set of solutions to a PBMod-constraint modulo $M_3 =$ lcm(M_1, M_2), one can find the set of solutions to two PBMod-constraints (modulo M_1 and M_2) and return their intersection.

Proposition 26 *Let Q be PB-constraint $\sum a_i l_i = b$, and $\mathtt{M} = \{M_1, \cdots, M_m\}$ be a set of m positive integers. The set of assignments satisfying Q is exactly the same as the set of assignments satisfying all the m PBMod-constraints, $Q[M_1], Q[M_2], \cdots, Q[M_m]$ if lcm$(M_1, \cdots, M_m) > \sum a_i$.*

Proof: This proposition can be proven by applying Lemmas 2 and 3. ■

Theorem 3 *Let Q denote the PB-constraint $\sum a_i l_i = b$ and $\mathtt{M} = \{M_1, \cdots, M_m\}$ be a set of positive integers such that, $lcm(M_1, \cdots, M_m) \geq \sum a_i$. Let the pair $\langle v_k, C_k \rangle$ be a translation for $Q[M_k]$. Then pair $\langle v, C \rangle$, where $C = \cup_k C_k \cup C'$ and C' is a set of clauses equivalent to $v \leftrightarrow (v_1 \wedge \cdots \wedge v_m)$, is a translation for Q.*

Proof: To prove this theorem, we must show that the set of clauses C is satisfiable and that, for every satisfying assignment τ for C, $\tau \models v$ iff $\tau \models Q$.

- In the next section, we describe how our translations for PBMod-constraints work, and prove that our proposed translations are such that set of clause C is always satisfiable.
- Let τ be a satisfying assignment for C such that $\tau \models v$. Based on the construction, $\tau \models v$ iff $\tau \models v_i$, for $1 \leq i \leq m$. Since $\langle v_k, C_k \rangle$ is a translation for $Q[M_k]$, we infer that $\tau \models Q[M_i]$, for all $i = 1, \cdots, m$. Proposition 26 shows that $\tau \models Q$.

■

Let Q be PB-constraint $\sum a_i l_i = b$ and $\mathtt{M^N} = \{2, \cdots, \lceil \log \sum a_i \rceil + 1\}$. Since lcm$(2, \cdots, k) \geq 2^{k-1}$, [37], set $\mathtt{M^N}$ can be used as the set of modulos for encoding Q.

Another candidate for set $\mathtt{M}$ is a subset of prime numbers. One can enumerate the prime numbers and add them to the set of modulos, $\mathtt{M^P}$, until their multiplication exceeds $S = \sum a_i$, i.e., to select $\mathtt{M^P}$ to be $\{2, 3, ..., P_m\}$. The next proposition provides an estimation of the size of set $\mathtt{M^P}$ as well as the maximum value in $\mathtt{M^P}$.

Proposition 27 *Let $\mathtt{M^P}$ be the set of primes less than or equal to P_m, where P_m is a prime number, such that $\prod_{p \in \mathtt{M^P}} p \geq S$.*

1. $m = |\mathtt{M^P}| = \theta(\frac{\ln S}{\ln \ln S})$.
2. $P_m < \ln S$.

Proof: Let $\mathtt{M^P}$ be the set of m first prime numbers, $\mathtt{M^P} = \{2, 3, \cdots, P_m\}$ and S be an integer.

Let $\pi(x)$ denote the number of primes less than or equal to x. We can bound the value of $\Pi_{p \in \mathtt{M^P}} p$, by:

$$\left(\frac{P_m}{e}\right)^{\pi(P_m) - \pi(P_m/e)} \leq \prod_{p \in \mathtt{M^P}} p \leq (P_m)^{\pi(P_m)} \tag{10.4}$$

Prime number theorem, [45], states that the number of primes less than or equal to an integer x, $\pi(x)$, satisfies the following:

$$\lim_{x \mapsto \infty} \frac{\pi(x)}{x/ln(x)} = 1. \tag{10.5}$$

Using the above formula, for large enough x, we rewrite (10.4), by replacing $\pi(x)$ with $x/\ln x$, as:

$$\left(\frac{P_m}{e}\right)^{P_m/\ln(P_m)-P_m/(e*\ln(P_m/e))} \leq \prod_{p\in \mathtt{M^P}} p \leq (P_m)^{P_m/\ln(P_m)} \leq e^{P_m}. \tag{10.6}$$

The lower bound for $\Pi_{p\in \mathtt{M^P}}$, for large enough P_m, is equal to:

$$\begin{aligned}&\left(\frac{P_m}{e}\right)^{P_m/\ln(P_m)-P_m/(e*\ln(P_m/e))}\\ &= (e)^{\frac{P_m*(\ln P_m-1)}{\ln P_m}-\frac{P_m}{e}} \geq (e)^{\frac{P_m}{2}-\frac{P_m}{e}}.\end{aligned} \tag{10.7}$$

From (10.6) and (10.7), we have

$$(e)^{\frac{P_m}{2}-\frac{P_m}{e}} \leq \prod_{p\in \mathtt{M^P}} p \leq e^{P_m} \tag{10.8}$$

$$\prod_{p\in \mathtt{M^P}} p \in e^{\theta(P_m)}. \tag{10.9}$$

The last equation, (10.9), states that the maximum value in $\mathtt{M^P}$ is $\theta(\ln S)$.
Now, by applying the prime number theorem once more, we see that:

$$m = |\mathtt{M^P}| \approx \frac{P_m}{\ln P_m} = \theta(\frac{\ln S}{\ln\ln S})$$

■

We can also use the following set of numbers as our set of modulos:

$$\mathtt{M^{PP}} = \{P_i^{n_i} : P_i \text{ is i-th prime number and } P_i^{n_i-1} \leq \lg S \leq P_i^{n_i}\}.$$

It is straightforward to see that $|\mathtt{M^{PP}}| \leq \frac{\lg S}{\lg\lg S}$ and $max_{M\in \mathtt{M^{PP}}} M \leq \lg S$.

It is worth mentioning that PB-constraint Q, in form (10.1), can be represented using $\theta(n \log a_{Max})$ bits where n is the number of literals (coefficients) in the constraint and a_{Max} is the maximum value of the coefficients. PBMod-constraint $Q[M]$ can be represented in $\theta(n \log M)$ bits where n is the number of literals in the constraint. So, if we determine a translation for $Q[M]$ which produces a CNF with $O(n^{k_1}M^{k_2})$, for some constants k_1 and k_2, clauses/variables, which is exponential in the size of representation of $Q[M]$, we can translate the PB-constraints into CNF using a polynomial number of variables (clauses, literals) with respect to the size of representation of the original PB-constraint. In the next section, we propose two such translations.

10.4 Encoding For Modular Pseudo-Boolean Constraints

Theorem 3, in the previous section, demonstrate an approach for constructing a translation for PB-constraints, using translations for PBMod-constraints. In this section, we describe how a translation for PBMod-constraints in the form (10.3) can be constructed. Remember that our ultimate goal is not to translate PBMod-constraints but to translate PB-constraints. For the sake of explanation, we assume all coefficients in the PBMod-constraint (a_i) are non-zero.

In this section, we present two encodings for PBMod-constraints and discuss how unit propagation performs on the CNF generated by each of them.

10.4.1 Dynamic Programming Based Translation (DP)

The translation presented here encodes PBMod-constraints of the form 10.3 using a Dynamic Programming approach. Using an appropriate set of clauses, C, we define the truth value of the auxiliary variables in $D = \{D^j_m : 0 \leq j \leq n, 0 \leq m \leq M\}$ such that in every total assignment τ satisfying C, τ maps variable D^j_m to *true* iff τ satisfies subproblem $\sum_{i=1}^{j} a_i l_i \equiv m (\text{mod M})$. The set of clauses C must be such that it guarantees the following assertions:

1. If both $D^{j-1}_{m-a_j}$ and l_j are *true*, D^j_m must be *true*.
2. If D^{j-1}_m is *true* and l_j is *false*, D^j_m must be *true*.
3. If D^j_m is *false* and l_j is *true*, $D^{j-1}_{m-a_j}$ must be *false*.
4. If both D^j_m and l_j are *false*, D^{j-1}_m must be *false*.
5. If both D^{j-1}_m and $D^{j-1}_{m-a_j}$ are *false*, D^j_m must be *false*;
6. If D^j_m is *true*, exactly one of $D^{j-1}_{m-a_j}$ and D^{j-1}_m must be *true*;

We must also add the following unary clauses to C:

1. The PBMod-constraint with no term in the left-hand side and zero in the right-hand size is always satisfiable: D^0_0
2. If $m \neq 0$, D^m_0 is *false*.

Proposition 28 *Let $D = \{D^j_m : 0 \leq j \leq n, 0 \leq m \leq M\}$. For every $j > 0$ and $0 \leq m \leq M$, we define $C^j_m = \{\{\neg D^{j-1}_{m-a_j}, \neg l_j, D^j_m\}, \{\neg D^{j-1}_m, l_j, D^j_m\}, \{\neg D^j_m, D^{j-1}_m, D^{j-1}_{m-a_j}\}, \{\neg D^j_m, \neg D^{j-1}_m, \neg D^{j-1}_{m-a_j}\}$ $\{\neg D^j_m, \neg l_j, \neg D^{j-1}_m\}, \{\neg D^j_m, \neg l_j, D^{j-1}_{m-a_j}\}, \{\neg D^m_j, l_j, \neg D^{j-1}_{m-a_j}\}, \{\neg D^m_j, l_j, D^{j-1}_m\}\}$. Then pair $\langle D^n_b, C\rangle$, where $C = \cup_{m,j} C^j_m \cup \{\{\neg D^0_m\} : m \neq 0\} \cup \{\{D^0_0\}\}$ is a translation for (10.3).*

Proof: To prove that the pair $\langle D^n_b, C\rangle$ is a translation for $\sum_{i=1}^{n} a_i l_i = b$, we first show that for every satisfying assignment τ for C, $\tau \models \sum_{i=1}^{n} a_i l_i = b$ iff $\tau \models D^n_b$. We also show that C is satisfiable.

By induction on n, we prove that the pair $\langle D^n_b, C\rangle$, for all $0 \leq b < M$, is a translation for PBMod-constraint $\sum a_i l_i = b$ (mod M):

The base cases are when $n = 0$:

- If $b = 0$, D^0_0 is always *true*, and the corresponding PBMod-constraint is always *true*.
- If $b \neq 0$, the PBMod-constraint $\sum_{i=1}^{0} a_i l_i = b$ is always unsatisfiable and so, the pair $\langle D^0_b, C\rangle$, where D^0_b is always *false*, is a translation for this constraint.

Inductive Step: We want to show that the pair $\langle D_b^n, C\rangle$, as described above, is a translation for $\sum_{i=1}^{n} a_i l_i = b$. By the induction hypothesis, $\langle D_{n-1}^b, C\rangle$ is a translation for $\sum_{i=1}^{n-1} a_i l_i = b$ and $\langle D_{n-1}^{b-a_n}, C\rangle$ is a translation for $\sum_{i=1}^{n-1} a_i l_i = b - a_n$, and by our construction, we have that C contains all the clauses in $C_m^{b'}$, $0 \leq m \leq n, 0 \leq b' < M$.

Let τ be a satisfying assignment for C such that $\tau[X] \models \sum_{i=1}^{n} a_i l_i = b$. We need to show that $\tau \models D_n^b$.

Since $\tau[X] \models \sum_{i=1}^{n} a_i l_i = b$, we have either $\tau[X] \models \sum_{i=1}^{n-1} a_i l_i = b$ and $\tau \models \neg l_n$ or $\tau[X] \models \sum_{i=1}^{n-1} a_i l_i = b - a_n$ and $\tau \models l_n$. By the induction hypothesis, we know that in every satisfying assignment τ for C, the truth value of D_{n-1}^b ($D_{n-1}^{b-a_n}$) is the same as the truth value of $\sum_{i=1}^{n-1} a_i l_i = b$ ($\sum_{i=1}^{n-1} a_i l_i = b - a_n$, respectively).

We have either $\tau \models D_{n-1}^{b-a_n}$ or $\tau \models D_{n-1}^b$. Assume that $\tau \models D_{n-1}^{b-a_n}$, so from the induction hypothesis, we know that $\sum_{i=1}^{n-1} a_i l_i = b - a_n$ and also from the fact that $\tau \models \sum a_i l_i = b$, we know that $\tau \models l_n$. Since $\tau \models C$ and C contains the clause $\{\neg D_{n-1}^{b-a_n}, \neg l_n, D_n^b\}$, we have $\tau \models D_b^n$. The proof for the other case, when $\tau \models D_b^{n-1}$, is similar.

To complete the proof, we still need to show that if τ is a satisfying assignment for C such that $\tau \models D_b^n$, then $\tau[X] \models \sum_{i=1}^{n} a_i l_i = b$.

From $\tau \models D_b^n$, $\{\neg D_b^n, D_b^{n-1}, D_{b-a_n}^{n-1}\} \in C$ and $\tau \models C$, we infer either $\tau \models D_{n-1}^b$ or $D_{n-1}^{b-a_n}$ but not both. We assume $\tau \models D_{n-1}^b$, and so, by the induction hypothesis, $\tau[X] \models \sum_{i=1}^{n-1} a_i l_i = b$. UP infers $\tau \models \neg D_{n-1}^{b-a_n}$, and using the clause $\{\neg D_b^n, \neg l_n, \neg D_b^{n-1}\} \in C$, UP infers that $\tau \models \neg l_n$. Since $\tau \models \sum_{i=1}^{n-1} a_i l_i = b$ and $\tau \models \neg l_n$, we have $\tau \models \sum_{i=1}^{n} a_i l_i = b$. The proof for the other case, where $\tau \models D_{b-a_n}^{n-1}$, is similar.

Instead of proving C is satisfiable, we prove the following which is a stronger assertion:

Given total assignment τ_n' to set of variables $X_n = \{var(l_i) \mid 1 \leq i \leq n\}$, there is a unique total assignment τ_n to $Y_n = X_n \cup \{D_b^i \mid 0 \leq b < M, 0 \leq i \leq n\}$ which satisfies C.

We use induction to prove this assertion:

Base case, when $n = 0$, is trivial, since the truth values of D_b^0, $0 \leq b < M$, are forced using unary clauses in C.

Inductive step: Let τ_n' be a total assignment to X_n and τ_{n-1}' be a restriction of τ_n' to $X_{n-1} = \{var(l_i) \mid 1 \leq i < n\}$. By the induction hypothesis, there is a unique total assignment τ_{n-1} to $Y_{n-1} = X_{n-1} \cup \{D_b^i \mid 0 \leq b < M, 0 \leq i < n\}$ which satisfies C.

Since $\tau_n' \models l_n$, we have $\tau_n \models l_n$. Also, since τ_n' and τ_{n-1}' agree on their assignments to variables in X_{n-1}, τ_{n-1} and τ_n also agree on their assignments to variables in Y_{n-1}.

Let's assume that $\sum_{i=1}^{n-1} a_i l_i$ evaluates, under τ_{n-1}', to b, and $\tau_n' \models \neg l_n$. Using the proof of the first part of the proposition, we have $\tau_{n-1} \models D_b^{n-1}$ and $\tau_{n-1} \models \neg D_{b'}^{n-1}$, where $b' \neq b$. Since we assumed $\tau_n' \models \neg l_n$, we need to show that the only expansion of τ_{n-1} to τ_n which satisfies all the clauses in C must satisfy $\tau_n \models b$ and $\tau_n \models \neg b'$, where $b' \neq b$.

1. $\tau_n \models D_b^n$: There is a clause in C asserting $\neg D_b^{n-1} \wedge \neg l_n \rightarrow D_{b-1}^n$ and since $\tau_n \models \neg D_b^{n-1} \wedge \neg l_n$, $\tau_n \models D_b^n$.
2. $\tau_n \models \neg D_{b'}^n$: There is a clause in C asserting $D_b^n \wedge \neg l_n \rightarrow D_b^{n-1}$ and since $\tau_n \models \neg D_b^{n-1}$ and $\tau \models \neg l_n$, τ cannot map D_b^n to *true*. So, τ must map D_b^n to *false*.

■

Proposition 29 *DP encoding produces a CNF with $|D| = O(nM)$ auxiliary variables, and $O(nM)$ clauses.*

Proof: DP encoding introduces an auxiliary variable for every $0 \leq i \leq n$ and $0 \leq b \leq M$. So, overall there are $O(nM)$ auxiliary variables. From Proposition 28, we know that the construction for DP encoding uses a constant number of clauses, for each $0 \leq i \leq n$ and $0 \leq b \leq M$, and therefore, overall we have $O(nM)$ clauses.

■

By applying standard dynamic programming techniques, we can avoid describing unnecessary D^j_m, and obtain a smaller CNF.

Binary Decision Diagrams (BDD) are standard tools for translating constraints to SAT. One can construct a BDD encoding for PBMod-constraints similar to the BDD encoding for PB-constraints described in [34]. Similar to what has been discussed in the previous Chapter, BDD introduces more auxiliary variables than DP encoding and unit propagation can infer more facts on the CNF generated by DP encoding.

To boost the performance of unit propagation on the CNF generated by DP encoding, we add the following set of clauses to C, defined in Proposition 28.

1. If $D^j_{m_1}$ is *true*, $D^j_{m_2}$ must be *false* $m_1 \neq m_2$, $1 \leq j \leq n$, i.e., $\{\neg D^j_{m_1}, \neg D^j_{m_2}\}$.
2. There is at least one m such that D^j_m is *true* $1 \leq j \leq n$, i.e., $\{D^j_m | m = 0, \cdots, M-1\}$.

Proposition 30 *DP encoding for PBMod-constraints is generalized arc consistent.*

Proof: To prove that a given encoding is generalized arc consistent, we must show that it supports both UP-detect inconsistent and UP-maintain GAC. Here, we only show the proof for DP encoding supports UP-detect inconsistency. The proof for the other part is similar. To simplify the proof, we assume PBMod-constraint is in the form $\sum a_i x_i \equiv b$ mod M, instead of $\sum a_i l_i \equiv b$ mod M.

Recall that encoding E supports UP-detect inconsistency if, for given PB-constraint Q on X in form $\sum_{i=1}^{n} a_i x_i \equiv b$ mod M, the pair $\langle v, C \rangle$, produced by encoding E has the following property:

For every (partial) assignment τ to variables in X, where all assignments expanding τ falsify Q, we have $UP(\tau, C) \models \neg v$.

We prove this assertion by induction on the number of literals in Q:

- The base cases are the constraints with no literal (coefficient): If the right-hand side of PBMod-constraint is 0, the constraint is a trivial (always *true*) otherwise, it is always *false*. The set of clauses generated for this kind of constraint contains M unary clauses, where M is the modulo in Q. Also, variable v is D^0_b, where b is the constant on the right-hand side of Q. So, UP can successfully infer the value of v.

- Inductive step: Let Q be in the form $\sum_{i=1}^{n+1} a_i x_i \equiv b$ mod M and also let τ be an assignment which cannot be expanded to a satisfying assignment for Q. There are three cases:

  1. τ does not set x_{n+1}: Knowing τ cannot be expanded to a satisfying assignment for Q means that τ cannot be expanded to a satisfying assignment for either $\sum_{i=1}^{n} a_i x_i = b$ or $\sum_{i=1}^{n} a_i x_i = b - a_{n+1}$. So, by the induction hypothesis, we have $UP(\tau, C) \models \neg D^n_b$ and $UP(\tau, C) \models \neg D^n_{b-a_{n+1}}$. From the construction, we know that C contains clause $\{D^n_b, D^n_{b-a_{n+1}}, \neg D^{n+1}_b\}$ and so, UP infers $\neg D^{n+1}_b$.

2. τ maps x_{n+1} to *true*: Since τ cannot be expanded to a satisfying assignment for Q and $\tau \models x_{n+1}$ mean that $\tau[\{x_1, \cdots, x_n\}]$ cannot be expanded to a satisfying assignment for $\sum_{i=1}^{n} a_i x_i = b - a_{n+1}$. So, by the induction hypothesis, we have $UP(\tau, C) \models \neg D^n_{b-a_{n+1}}$. From the construction, we know that C contains clause $\{D^n_{b-a_{n+1}}, \neg x_{n+1}, \neg D^{n+1}_b\}$ and so, UP infers $\neg D^{n+1}_b$.
3. τ maps x_{n+1} to *false*: The proof for this case is similar to that of the previous case.

■

10.4.2 Divide and Conquer Based Translation (DC)

The translation presented here resembles a Divide and Conquer approach. Using appropriate set of clauses C, we define the truth values of the auxiliary variables in $D = \{D^a_{s,l} : 0 \leq a \leq M, 0 \leq s \leq n, 0 \leq l \leq n\}$ such that in every total assignment τ satisfying C, τ maps $D^a_{s,l}$ to *true* iff τ satisfies subproblem $\sum_{i=s}^{s+l-1} a_i l_i \equiv a \pmod{\mathrm{M}}$.

Let $D_{s,l} = \{D^a_{s,l} : 0 \leq a < M\}$. The set of clauses C must be such that it guarantees the following assertions among the truth values of variables in sets $D_{s,l}$, $D_{s,\frac{l}{2}}$ and $D_{s+\frac{l}{2},\frac{l}{2}}$:

- If both $D^{m_1}_{s,\frac{l}{2}}$ and $D^{m-m_1}_{s+\frac{l}{2},\frac{l}{2}}$ are *true*, then $D^m_{s,l}$ must be *true*, where $0 \leq m, m_1 < M$, i.e., $(D^{m_1}_{s,\frac{l}{2}} \wedge D^{m-m_1}_{s+\frac{l}{2},\frac{l}{2}}) \rightarrow D^m_{s,l}$.
- If $D^{m_1}_{s,l}$ is *true*, $D^{m_2}_{s,l}$ must be *false*, where $0 \leq m_1 < m_2 < M$, i.e., $D^{m_1}_{s,l} \rightarrow \neg D^{m_2}_{s,l}$.
- There is at least one m, $0 \leq m < M$, such that $D^m_{s,l}$ is *true*, i.e., $\bigcup_m D^m_{s,l}$.

For the base cases, when $l = 1$, we use the followings assertions.

1. $D^{a_s}_{s,1}$ is equivalent to x_s.
2. $D^0_{s,1}$ is equivalent to $\neg x_s$.
3. $D^m_{s,1}$, $m \neq a_s$, is *false*.

Proposition 31 *Let* $D = \{D^m_{s,l} \mid 0 \leq m \leq M, 0 \leq s, l \leq n\}$ *and* $C = \bigcup_{m,s,l} C^m_{s,l} \cup \bigcup_{m_1,m_2,s,l} C^{m_1,m_2}_{s,l} \cup \bigcup_{s,l} C_{s,l}$ *, where*

1. $C^m_{s,l}$ *(*$l > 1$*) is* $\{\{\neg D^{m_1}_{s,\frac{l}{2}}, \neg D^{m-m_1}_{s+\frac{l}{2},\frac{l}{2}}, D^m_{s,l}\} \mid 0 \leq m_1 < M\}$*, which expresses* $(D^{m_1}_{s,\frac{l}{2}} \wedge D^{m-m_1}_{s+\frac{l}{2},\frac{l}{2}}) \rightarrow D^m_{s,l}$*;*
2. $C^m_{s,l}$ *(*$l = 1$*) is* $\{\{\neg x_s, D^{a_s}_{s,1}\}, \{x_s, \neg D^{a_s}_{s,1}\} \mid 1 \leq s \leq n\} \cup \{\{\neg D^m_{s,1} \mid 1 \leq s \leq n, m \neq a_s\}$*, which expresses* $D^{a_s}_{s,1}$ *is equivalent to* x_s *and* $D^m_{s,1}$ *is false;*
3. $C^{m_1,m_2}_{s,l}$ *is* $\{\{\neg D^{m_1}_{s,l}, \neg D^{m_2}_{s,l}\}\}$*, which expresses* $D^{m_1}_{s,l} \rightarrow \neg D^{m_2}_{s,l}$*;*
4. $C_{s,l}$ *is* $\{\{D^m_{s,l} \mid 0 \leq m < M\}\}$*, which expresses* $\bigcup_m D^m_{s,l}$*.*

Then pair $\langle D^b_{1,n}, C\rangle$ *is a translation for (10.3).*

Proof: The proof for this proposition is similar to that of Proposition 28. ■

Similar to DP encoding, DC encoding is also an arc consistent encoding for PBMod-constraints.

Proposition 32 *DC encoding for PBMod-constraints is generalized arc consistent.*

Proof: The proof for this proposition is similar to that of Proposition 30. ■

The way we define the set of auxiliary variables D requires $|D|$ to be n^2M. Using the standard top-down implementation technique for divide and conquer algorithms reduces the number of variables in D to $O(nM)$. This implementation can be seen in the following construction:

1. Start by producing set of clauses $C_{1,n}$,
2. To produce $C_{s,l}$, we need to generate M auxiliary variables, $D^0_{s,l}, \cdots, D^{M-1}_{s,l}$, and produce sets of clauses $C_{s,\frac{l}{2}}$ and $C_{s+\frac{l}{2},\frac{l}{2}}$.

Proposition 33 *The above top-down implementation for DC encoding, produces a CNF with $|D| = O(nM)$ auxiliary variables, and $O(nM^2)$ clauses.*

Proof: Let $T_{s,l}$ be the number of auxiliary variables generated to describe $C_{s,l}$. The number of auxiliary variables generated by the top-down implementation of DC encoding can be described using the following recurrence formula:

$$T_{s,l} = \begin{cases} M & \text{if } l = 1; \\ M + T_{s,\frac{l}{2}} + T_{s+\frac{l}{2},\frac{l}{2}} & \text{otherwise.} \end{cases}$$

By solving the recurrence formula, we get:

$$T_{1,n} = \theta(nM).$$

In the CNF generated by DC encoding, each propositional variable $D^m_{s,l} \in D$ occurs in at most $\theta(M)$ clauses. Therefore the number of clauses is $\theta(nM^2)$.

■

Table 10.1 summarizes the number of auxiliary variables, the number of clauses and the depth of corresponding circuit in the CNF obtained from the DP and DC encodings described above for translating PBMod-constraint $\sum a_i l_i \equiv b \pmod{\text{M}}$.

Encoder	# of Aux. Vars.	# of Clauses
DP	$O(nM)$	$O(nM)$
DC	$O(nM)$	$O(nM^2)$

Table 10.1: Summary of the size of different encodings for $\sum a_i l_i \equiv b \pmod{\text{M}}$.

In the previous section we described two candidates for modulo sets, $\mathtt{M}^{\mathtt{P}}$ and $\mathtt{M}^{\mathtt{PP}}$. In this section, we explained two encodings for translating PBMod constraints to SAT, DP and DC encodings. So, we can construct four different encodings for translating PB constraints to SAT:

- Prime.DP: Translation constructed using $\mathtt{M}^{\mathtt{P}}$ as the set of modulo and DP encoding as the PBMod encoder;
- Prime.DC: Translation constructed using $\mathtt{M}^{\mathtt{P}}$ as the set of modulo and DC encoding as the PBMod encoder;

- PPower.DP: Translation constructed using $\mathtt{M}^{\mathtt{PP}}$ as the set of modulo and DP encoding as the PBMod encoder;
- PPower.DC: Translation constructed using $\mathtt{M}^{\mathtt{PP}}$ as the set of modulo and DC encoding as the PBMod encoder.

The following proposition bounds the number of auxiliary variables and number of clauses in the CNF generated by each of these four encodings.

Proposition 34 *Let Q be a PB-constraint in from $\sum a_i l_i = b$, and $S = \sum a_i$. The following Table summarizes the number of auxiliary variables/clauses and the depth of the corresponding circuit in the CNF generated by each of the Prime.DP/Prime.DC/PPower.DP/PPower.DC encodings for Q.*

Encoding	# of Vars.	# of Clauses
Prime.DP	$O(n\frac{\ln^2(S)}{\ln\ln S})$	$O(\frac{n\ln^2(S)}{\ln\ln(S)})$
Prime.DC	$O(n\frac{\ln^2(S)}{\ln\ln(S)})$	$O(n\frac{\ln^3(S)}{\ln\ln(S)})$
PPower.DP	$O(n\frac{\log^2(S)}{\log\log S})$	$O(\frac{n\log^2(S)}{\log\log(S)})$
PPower.DC	$O(n\frac{\log^2(S)}{\log\log(S)})$	$O(n\left(\frac{\log(S)}{\log\log(S)}\right)^2)$

Proof: The number of variables/clauses in the CNF generated by each of these four encodings is equal to the summation of the number of variables/clauses generated by PBMod encoder, for different modulos. Here we only present the proof for the number of variables in Prime.DP encoding.

From Proposition 29, we know the number of variables in the CNF generated by DP encoding for PBMod constraint $Q[M]$ is $\theta(nM)$. We know that the set of modulos, M^P, contains the first $|M^P|$ prime numbers. So, the number of variables in the CNF generated by Prime.DP is:

$$\sum_{i=1}^{|M^P|} nP_i = n\sum_{i=1}^{|M^P|} P_i, \tag{10.10}$$

where P_i is the i-th prime number. Let P_m be the maximum integer in M^P.

For $k \geq 1$, we have the following relation for the sum of prime powers, [60]:

$$\sum_{p\leq x} p^k = \frac{(1+o(1))x^{k+1}}{(1+k)\log x}. \tag{10.11}$$

Using Proposition 27 and the above equation, by replacing with x with P_m, we rewrite (10.10) as

$$n\sum_{i=1}^{|M^P|} P_i = \frac{(1+o(1))P_m^2}{2\log P_m}. \tag{10.12}$$

By replacing P_m with the upperbound for P_m, we get

$$n \sum_{i=1}^{|M^P|} P_i = \frac{(n + o(n))(\ln S)^2}{2 \log \ln S}, \tag{10.13}$$

where $S = \sum a_i$.

■

10.5 Previous Work

The existence of a polynomial size arc consistent encoding for PB-consistent in the form $\sum a_i x_i < b$ was an open question until very recently. Bailleux et al. developed an arc consistent polynomial size translation for these constraints [13]. Although all kinds of PB-constraints can be written as the conjunction of at most two constraints in the form $\sum a_i l_i < b$, the arc consistency is not preserved for PB-constraints in the form $\sum a_i l_i = b$. Moreover, in Section 10.6, we prove there cannot be a polynomial size arc consistent encoding for all possible PB-constraints in form $\sum a_i l_i = b$ unless P=co-NP.

There is an arc consistent encoding based on Binary Decision Diagrams. This encoding creates exponential size CNF and is not practical when the integers in the PB-constraints are large. Here we briefly review this encoding.

Translation through BDD

This approach is similar to the dynamic programming solution for solving the Subset Sum problem. For every possible pair i and j where $0 \leq i \leq n$, $0 \leq j \leq b$, a fresh Tseitin variable is introduced, D^i_j, and using appropriate clauses the relation among the truth values of D^i_j, x_i, D^{i-1}_j and $D^{i-1}_{j-a_i}$ is described.

$$D^j_i = \begin{cases} \top & \text{if } i \text{ and } j \text{ are both zero;} \\ \bot & i = 0 \text{ and } j > 0; \\ (D^{i-1}_{j-a_i} \wedge x_i) \vee (D^{i-1}_j \wedge \neg x_i) & Otherwise. \end{cases}$$

Describing the truth value of variables in a top-down manner, as proposed by [12], usually generate fewer number of Tseitin variables and smaller CNF than the bottom-up procedure. Translation through BDD is arc consistent but may produce an exponential size CNF with respect to the input size.

There are other encodings which produce polynomial size CNFs. In the rest of this section, we review some of these encodings.

Binary Encoding (Bin)

Every circuit can be translated into CNF, and so the binary adders can be described using a series of clauses. The main idea in this approach is to use binary encoding of integers and the fact that setting x_i to *false* is the same as setting a_i to zero. Every coefficient in a PB-constraint, a_i, is represented

as a tuple of bits $\langle c_i^1 \wedge x_i, \cdots, c_i^k \wedge x_i \rangle$ and each of these tuples is fed into an adder-network. The output of the adder-network is compared with the binary representation of b.

The size of CNF generated using this encoding is polynomial with respect to the size of input, but unit propagation performs poorly on the produced CNF.

Translation Through Totalizer

In [13], Bailleux et al. described an encoding for PB-constraints in the form $Q : \sum a_i x_i < b$ which fully supports arc consistency and produces a polynomial size CNF. In their context, setting a variable from X to false never makes the constraint inconsistent, i.e., the formula $\neg Q$ is *a monotone formula* [9].

Bailleux et al. used a construction, called *polynomial watchdog*. A polynomial watchdog associated with the constraint Q on variables X is a CNF formula, $PW(Q)$, such that for every partial assignment to the input variables, X, that violates the constraint Q, the unit propagation applied to $PW(Q)$ infers the value *true* for the output variable of $PW(Q)$.

If constraint $Q : \sum a_i x_i < b$ is not satisfiable under a partial assignment, the sum of the coefficients of variables which are set to true under the current partial assignment must be greater than or equal to b. The variable x_k is forced to be false under the current assignment iff $Q_k : \sum_{i \neq k} a_i x_i < b - a_i$, is not consistent. Global polynomial watchdog (GPW) and Local polynomial watchdogs (LPW) are used to enable UP to carry out these kinds of inferences. The following can be used as an encoding for PB-constraint Q.

$$F = \neg GPW(Q) \wedge \bigwedge LPW(Q_k) \Rightarrow (\neg x_k).$$

With access to an encoding for PB-constraints in the form $Q : \sum a'_i l_i = b'$, one can build an encoding for constraint $Q' : \sum a_i l_i < b$ using Observation 7.

Observation 7 *The set of solutions to PB-constraint (1) is the same as the intersection of the sets of solutions to PB-constraints (2) and (3).*

1. $\sum a_i l_i = b$;
2. $\sum a_i l_i < b + 1$;
3. $\sum a_i \neg l_i < \sum a_i + 1 - b$.

There are normalized PB-constraints for which totalizer based translation is not arc consistent but our encoding is. We characterized these instances in Section 10.6.

Translation Through Network of Sorters (SN)

A sorting network is a circuit with n input wires and n output wires consisting of a set of comparators with two input wires and two output wires. Each output of a comparator is used as an input to another comparator, except those used as the output wires of the sorting network.

In this translation, a mixed-base, $B = \langle B_1, \cdots, B_k \rangle$, was selected. Each coefficient, a_i, is represented using a vector of size k, $\langle c_i^1, \cdots, c_i^k \rangle$ such that $0 \leq c_i^j < B_j$ and

$$a_i = \sum_{j=1}^{k} c_i^j \prod_{k=1}^{j-1} B_k.$$

Table 10.2: Size of Translations ($a_{Max} = Max\{a_i\}$)

	# of Auxiliary Vars.	# of Clauses
BDD	$O(n^2 a_{Max})$	$O(n^2 a_{Max})$
Totalizer	$O(n^2 \log n \log a_{Max})$	$O(n^3 \log n \log a_{Max})$
Bin	$O(n \log a_{Max})$	$O(n \log a_{Max})$
SN	$O(n \log a_{Max} \log^2 \log a_{Max})$	$O(n \log a_{Max} \log^2 \log a_{Max})$
Prime.DP	$O(n(\log n + \log a_{Max})^2)$	$O(n(\log n + \log a_{Max})^2)$

Then each digit, c_i^j, is represented using B_j bits (in unary encoding), and k sorting networks are used to implement an adder-circuit which computes the summation of $(a_i \wedge x_i)$ for $1 \leq i \leq n$. One can find more details about the translation using a network of sorters in [34].

The size of the CNF generated using this encoding is polynomial with respect to the size of the input. This encoding is arc consistent if all the coefficients are one. This special class of PB-constraints is called *Cardinality Constraint* in the SAT community. There are some well-known encodings for cardinality constraints which are arc consistent and produce smaller CNFs [6].

10.5.1 Summary

Table 10.2 summarizes the number of auxiliary variables, clauses, and literals produced by each approach in the translation of $a_1 l_1 + \cdots + a_n l_n = b$.

BDD encoding is the only encoding which is arc consistent for the PB-constraint with comparison operator "=". This encoding may produce exponential size CNF.

We show in Section 10.6 that Totalizer encoding is not arc consistent for all constraints whose comparison operator is "=". The translation using sorting networks has a reasonable size but it is arc consistent if all the coefficients are equal to one (The authors in [34] demonstrated a *necessary condition* for arc consistency). As proved in Proposition 34, all of the four encodings, obtained using either $\mathtt{M}^{\mathtt{P}}$ or $\mathtt{M}^{\mathtt{PP}}$ as the set of modulos and DP or DC encodings for PBMod-constraints, produce a polynomial size CNF. In the next section, we show that the number of instances for which the CNF obtained by Prime.DP encoding is arc consistent is much greater than that of sorting networks. there are many instances for which our encoding is arc consistent while totalizer-based encoding is not.

In summary, Totalizer-based encoding, Sorting Network encoding and our encoding produce polynomial size translations for PB-constraint in the form $\sum a_i l_i = b$, each of which is arc consistent for a certain subset of all possible PB-constraints.

10.6 Performance of Unit Propagation

The encodings have been completely described and now, we are ready to study the properties of the proposed encodings. In this section, we show that there cannot be an encoding for PB-constraints in form $\sum a_i l_i = b$ which always produces a polynomial size arc consistent CNF unless P=co-NP.

We also study the arc consistency of our encoding and discuss why one can expect the proposed encodings to perform well.

In this section, we follow the notation introduced in Subsections 8.2.1 and 9.2.3. We also use the concepts of UP-detect inconsistency and UP-maintains GAC, described in Section 10.2.

10.6.1 Hardness Result

Here, we show that it is not very likely to have an arc consistent encoding which always produces a polynomial size CNF. We show this by proving if there is such an encoding, we can solve an instance of a hard problem in polynomial time.

Theorem 4 *There does not exist a UP-detectable encoding which always produces a polynomial size CNF unless P= co-NP. There does not exist a UP-maintainable encoding which always produces a polynomial size CNF unless P= co-NP.*

Proof: Given a set of integers $A = \{a_1, \cdots, a_n\}$ and an integer b, the Subset Sum problem asks if there exists non-empty subset of A whose sum of elements is equal to b. This problem is a well-known NP-complete problem, and we do not expect this problem to be solvable in polytime. We use the Subset Sum problem to prove this theorem.

Unit propagation completes its execution on a set of clauses either by reporting inconsistency or eliminating some variables from the input CNF. The worst-case running time of unit propagation is polynomial with respect to the number of literals in the input CNF.

An instance of the Subset Sum problem, defined by set $A = \{a_1, \cdots, a_n\}$ and integer b, can be represented as the following PB-constraint Q:

$$a_1x_1 + \cdots + a_nx_n = b.$$

Now, assume there exists an encoding whose resulting CNF is UP-detectable for all PB-constraints in the form $\sum a_i l_i = b$. Let's call this encoding E. Based on the definition of UP-detectability, E translates Q to $\langle v, C\rangle$ such that Q is unsatisfiable iff UP detects inconsistency when it runs on $\{v\} \cup C$.

The fact that UP can infer inconsistency on $\{v\} \cup \{C\}$ in polynomial time with respect to the number of literals in $\{v\} \cup C$ implies that if C has a polynomial size with respect to the size of the representation of Q, deciding whether the answer to a Subset Sum instance is "No" would be easy. Therefore, either there are PB-constraints whose corresponding CNFs are not of polynomial size or co-NP=P.

Assume we have a UP-maintainable encoding for PB-constraints. If PB-constraint Q on X has a unique solution, UP, given the CNF produced by UP-maintainable encoding, can infer the values of all Boolean variables in X. Also if UP fails to infer the values of all variables in X, know Q does not have a unique answer (it may be unsatisfiable or has more than one satisfying assignments). Now, we prove the second part of the theorem.

The Unique SAT problem (USAT) is a variant of the Satisfiability problem. USAT asks whether a given set of clauses has exactly one solution [18]. It is already known that USAT belongs to the complexity class D^P and it is co-NP-hard [57]. In the rest of our proof, we reduce an instance of

USAT problem, C, to an instance PB-constraint problem, Q, such that C has a unique satisfying assignment iff Q has a unique satisfying assignment.

Let $C = \{C_1, \cdots, C_m\}$ be a set of clauses on $X = \{x_1, \cdots, x_n\}$. For the sake of explanation, we assume every clause in C has exactly three literals. Let C_i be $\{l_i^1, l_i^2, l_i^3\}$. Then, we define PB-constraint Q as follows:

$$\sum_{i=1}^{m} 7^i l_i^1 + 7^i l_i^2 + 7^i l_i^3 + 7^i y_i + 2 \times 7^i z_i = \sum_{i=1}^{m} 4 \times 7^i, \tag{10.14}$$

where y_i and z_i are fresh Boolean variables, for $1 \leq i \leq m$.

It is straightforward to see that the number of satisfying assignment for C is the same as the number of satisfying assignment for Q. Therefore C has a unique solution iff Q has a unique solution.

PB-constraint Q is not in the canonical form, defined in Section 10.2, but it can be converted to a canonical PB-constraint using the procedure described in this chapter.

Let E be a UP-maintainable encoding for PB-constraints. Given a USAT instance C, we can construct PB-constraint Q in polynomial time. Let pair $\langle v, C' \rangle$ be the translation obtained by applying encoding E on constraint Q. UP on the set of clauses $C' \cup \{v\}$ infers the values of all variables $x \in X$ iff C has a unique solution.

Therefore either there are PB-constraints whose corresponding CNFs do not have polynomial size or we have a polynomial time algorithm which is able to solve co-NP-hard problem.

■

Throughout the rest of this section, we assume we are given a PB-constraint, Q, $\sum a_i l_i = b$, and a translation for it, $\langle v, C \rangle$. Also, let $Q_1, \cdots, Q_m$ be the PBMod-constraints generated during the translation process and $\langle v_i, C_i \rangle$ be a translation for Q_i. Also, let $Ans = \{\tau_1, \cdots, \tau_r\}$ be the set of all satisfying assignments to Q.

10.6.2 UP for Proposed Encodings

Although there is no arc consistent encoding for PB-constraints with "=", both DP-based and DC-based encodings for PBMod-constraints are arc consistent encodings.

In the rest of this section we study the cases for which we expect SAT solvers to perform well on the output of our encoding. Let Q be a PB-constraint on X, τ be a partial assignment and $Ans(\tau)$ be the set of total assignments, to X, satisfying Q and extending $\tau[X]$. There are two situations in which UP is able to infer the values of the input variables.

1. Unit Propagation Detects Inconsistency: One can infer that the current partial assignment, τ, cannot satisfy Q by knowing $Ans(\tau) = \emptyset$. If at least one of the m PBMod-constraints is inconsistent with the current partial assignment, UP can infer inconsistency on the CNF generated by our proposed encodings for PBMod-constraints.
   Recall that there are partial assignments and PB-constraints such that although $Ans(\tau) = \emptyset$, each of the m PBMod-constraints has a non-empty set of satisfying assignments (but the intersection of these sets is empty).
2. Unit Propagation Infers the value for an Input Variable: One can infer that the value of input variable x_k is *true/false* if x_k takes the same value in all the satisfying assignments of Q. For

this class of constraints, UP is able to infer the value of x_k, too.

If there exists a PBMod-constraint for which all its satisfying assignments, which extend τ, have mapped x_k to the same value, UP can infer the value of x_k.

These two cases are illustrated in the following example.

Example 35 *Let $Q(X)$ be $x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 = 12$.*

1. *Let τ, the current partial assignment, be $\{\neg x_2, \neg x_4\}$ and $M = 5$. There is no total assignment satisfying $1x_1 + 3x_3 + 0x_5 \equiv 2$ (mod 5).*
2. *Let τ, the current partial assignment, be $\{\neg x_3, \neg x_5\}$ and $M = 2$. There are four total assignments extending τ and satisfying PBMod-constraint $1x_1 + 0x_2 + 0x_4 \equiv 0$ (mod 2). In all of them, x_1 is mapped to false.*

A special case of the second situation is when UP detects the values of all $x \in X$, given the current partial assignment. In the rest of this section we estimate the number of PB-constraints for which UP can solve the problem. More precisely, we give a lower bound on the number of PB-constraints for which UP detects inconsistency or UP expands an empty assignment to a total assignment.

Proposition 35 *Let $R(n)$ be a polynomial in n such that $R(n) > 2n$ and $\mathcal{Q}$ the set of all constraints in form $\sum a_1 l_1 + \cdots + a_n l_n = b$, where $1 \leq a_i \leq U = 2^{R(n)}$ and $1 \leq b \leq n * U$. There are $2^{\Omega(nR(n))}$ many constraints in $\mathcal{Q}$, such that UP can solve the generated CNF using our proposed method in polynomial time.*

Proof: To simplify the analysis, we use the same set of prime modulos $\mathtt{M}^{\mathtt{P}} = \{P_1 = 2, \cdots, P_m = \theta(R(n)) > 2n\}$ for all constraints.

Consider the following PBMod-constraints:

$$1x_1 + \cdots + 1x_{n-1} + 1x_n = n + 1(\text{ mod } P_m) \tag{10.15}$$

$$1x_1 + \cdots + 1x_{n-1} + 1x_n = n(\text{ mod } P_m) \tag{10.16}$$

PBMod-constraint (10.15) does not have any solution and PBMod-constraint (10.16) has exactly one solution (in which all x_i are *true*). Proposition 30 tells us that UP can infer inconsistency given a translation obtained by DP-based encoding for (10.15), even if the current assignment is empty. Also, using Proposition 30, we know UP expands the empty assignment to an assignment mapping all x_i to *true* on a translation for (10.16) obtained by DP-based encoding.

The *Chinese Remainder Theorem* [32] implies that there are $(U/P_m)^{n+1} = 2^{(n+1)R(n)}/R(n)^{n+1}$ different PB-constraints in the form $\sum a_1 l_i = b$ such that their corresponding PBMod-constraints, where the modulo is P_m, are the same as (10.15). The same argument can be applied on PBMod-constraints in form (10.16).

Since UP can solve a PB-constraint in the form of Equations 10.15 or 10.16, and knowing that UP runs in linear time with respect to the number of clauses/variables in the CNF, there are at least $2^{(n+1)R(n)}/R(n)^{n+1}$ PB-constraints which can be solved, using the proposed translation, in polynomial time with respect to the size of the representation of the input PB-constraints.

■

10.6.3 UP for Sorting Network-based Encoding

Here, we show that there are more instances for which our encoding maintains arc consistency than Sorting Network encoding.

It is stated in [34]: "Unfortunately, arc consistency is broken by the duplication of inputs, both to the same sorter and between sorters."

As described in Section 10.5, in Sorting Network encoding, a multi-base $B = \langle B_1, \cdots, B_m \rangle$ is fixed. To avoid duplication between sorters, each coefficient, a_i, should have a single non-zero digit in their multi-base B-representation. To avoid duplication in the same sorter, the non-zero digit should be exactly 1. So each coefficient can take m different values, based on the position of its non-zero digit. There are n coefficients, so there are at most nUm^n different instances which are arc consistent, among all nU^{n+1} possible PB-constraints, where U is the maximum value each coefficient can have. Having $B_i \geq 2$ implies that $m \leq \log U$.

10.6.4 UP for Totalizer-based Encoding

In [13], the authors claimed that the totalizer-based encoding is a polynomial size CNF encoding such that arc consistency is maintained through unit propagation for all PB-constraints in the following form:

$$\sum a_i l_i \{=, >, \geq, <, \leq\} b.$$

Although the totalizer-based encoding is arc consistent for the PB-constraints in the forms $\sum a_i l_i \{>, \geq, <, \leq\}$, it does not produce an arc consistent translation for some PB-constraints in the form $\sum a_i l_i = b$.

In the totalizer-based encoding, PB-constraint $\sum a_i l_i = b$ is converted to the following two constraints:

$$\sum a_i l_i < b + 1;$$
$$\sum a_i \neg l_i < \sum a_i + 1 - b.$$

Proposition 36 *There are PB-constraints for which the totalizer-based encoding is not arc consistent.*

Proof: We have constructed an explicit counter example. Consider the following PB-constraint Q:

$$3x_1 + 3x_2 + 4x_3 = 7.$$

In all solutions for Q, x_3 should be mapped to *true*, and hence an arc consistent encoding must be able to infer the value of x_3 from an empty assignment. Now consider the following two constraints:

$$Q_1 : 3x_1 + 3x_2 + 4x_3 < 8,$$

$$Q_2 : 3\neg x_1 + 3\neg x_2 + 4\neg x_3 < 10 - 6 = 4.$$

Let $\langle v_1, C_1 \rangle$ and $\langle v_2, C_2 \rangle$ be translations obtained from the totalizer-based encoding for Q_1 and Q_2, respectively. UP does not infer the value of any x_i from $\{v_1\} \cup C_1 \cup \{c_2\} \cup C_2$. ■

In fact, the translation produced by the totalizer-based encoding is not arc consistent for almost all PB-constraints which have a PBMod-constraint in the form 10.15 or 10.16.

We summarize the discussion above in the following two observations.

Observation 8 *Let $R(n)$ be a polynomial in n such that $R(n) > 2n$ and Q the set of constraints in the form $\sum a_1 l_1 + \cdots + a_n l_n = b$, where $1 \leq a_i \leq U = 2^{R(n)}$ and $1 \leq b \leq n * U$. There are at most $(\log U)^n$ instances where the CNF produced by Sorting Network encoding maintains arc consistency, while this number for our encoding is at least $(U/\log(U)))^n$. We almost always have $2^{R(n)}/R(n) \gg R(n)$.*

Observation 9 *There is a family of PB-constraints whose translation through the totalizer-based encoding is not arc consistent but the translation obtained by our encoding is arc consistent.*

10.7 Experimental Evaluation

In this section, we compare the performance of our proposed encodings and two of the encodings described in the previous section. Remember that our encoders have two separate parts; a modulo selection part and a PBMod-constraint encoder part. In this section, we selected the following configurations: Prime as the set of modulos and DP encoding as PBMod-constraints encoders (Prime.DP), Prime as the set of modulos with DC encoding as PBMod-constraint encoder (Prime.DC). We used CryptoMiniSAT as the SAT solver for our encodings, as it performed better than MiniSAT, in our initial benchmarking experiments.

To evaluate the performance of these configurations, we used the Number Partitioning Problem (NPP). Given a set of integers $S = \{a_1, \cdots, a_n\}$, NPP asks whether there is a subset of S such that the summation of its members is exactly $\sum a_i/2$. Following [43], we generated 100 random instances for NPP, for a given n and L as follows:

Create set $S = \{a_1, \cdots, a_n\}$ such that each of a_i is selected independently at random from $[0, \cdots, 2^L - 1]$.

We selected the NPP problem as our benchmark since this problem allows us to change both the number of variables and the size of integers (coefficients) in the problem. Intuitively speaking, by changing the number of variables we change the size of the search space, and by changing the size of coefficients we change the size of the problem description.

We ran each instance on our two configurations and also on the three other encodings; Sorting Network-based encoding (SN), Binary Adder Encoding (BADD) [34] (both provided by MiniSAT+ [68]) and also another SAT-based PB-solver called npSolver [53]. Both MiniSAT+ and npSolver showed good performances in one or more categories of PB-competitions.

All running times reported here are the average total running times (the average over the result of summation of times spent to generate CNF formulas and time spent to solve the CNF formulas). We also tried to run the experiments with a BDD encoder, but as the CNF produced by BDD encoder is exponentially large, SAT solvers failed to solve medium and large size instances.

Before we describe the result of experiments, we discuss some properties of the Number Partitioning problem.

10.7.1 Number Partitioning Problem

The Number Partitioning problem is an NP-Complete problem which can also be seen as a special case of Subset Sum problem. In the SAT context, an instance of the NPP can be rewritten as a PB-constraint whose comparison operator is "=". Neither this problem nor the Subset Sum problem has received much attention from the SAT community.

The size of an instance of NPP, where set S has n elements and a_{Max} is the maximum absolute value in S, is $\theta(n * \log(a_{Max})) + n)$. It is known that if the value of a_{Max} is polynomial with respect to n, the standard dynamic programming approach can solve this problem in time $O(na_{ax})$, which is polynomial time with respect to the instance size. If a_{Max} is too large, $2^{Omega(2^{\theta(n)})}$, the naive algorithm, which generates all the 2^n subsets of S, works in polynomial time with respect to the instance size. The hard instances for this problem are those in which a_{Max} is neither too small nor too large with respect to n.

In [19], Borgs et al. defined $k = L/n$ and showed that NPP has a phase transition at $k = 1$: for $k < 1$, there are many perfect partitions with probability tending to 1 as $n \mapsto \infty$, while for $k > 1$, there are no perfect partitions with probability tending to 1 as $n \mapsto \infty$.

10.7.2 Experiments

All the experiments were performed on a Linux cluster (Intel(R) Xeon(R) 2.66GHz). We set the time limit to be 10 minutes. During our experiments, we noticed that the sorting network encoding in MiniSAT+ incorrectly announces some unsatisfiable instances to be satisfiable, an example of which is the following constraint.

$$5x_1 + 7x_2 + 1x_3 + 5x_4 = 9.$$

We did not investigate the reason for this issue in the source code of MiniSAT+, and all the reported timings are calculated using the broken code. We checked the solution generated by sorting network-based encoding of MiniSAT+ on the satisfiable instances, and they were indeed a correct solution. Also, for many of unsatisfiable instances, the sorting network-based encoding in MiniSAT+ reported unsatisfiability. Based on our observations, we believe that there are some rare corner cases which are missed in the implementation of sorting network-based encoding, in MiniSAT+. We should mention that, based on our experiments, the binary adder-based encoding implemented by MiniSAT+ is sound.

The results of experiments in this section suggest that Sorting Network-based encoding performs much worse than the other encodings. The relative performance of the encodings, we studied in this section, would not change unless the bug in MiniSAT+, for Sorting Network-based encoding, causes a huge slow-down (which is very unlikely).

In our experiments, we generated 100 instances for $3 \leq n \leq 30$ and $L \in \{3, \cdots, 2 * n\}$. We say a solver wins on a set of instances if it solves more instances than the others and in the case of a tie, we decide the winner by looking at the average running time. The instances on which each solver performed the best are plotted on Figure 10.1. As the Sorting Network-based solver was never a winner in any of the sets, it did not appear in the graph.

One can observe the following patterns from the data presented in Figure 10.1:

1. For $n < 15$, all solvers successfully solve all instances.

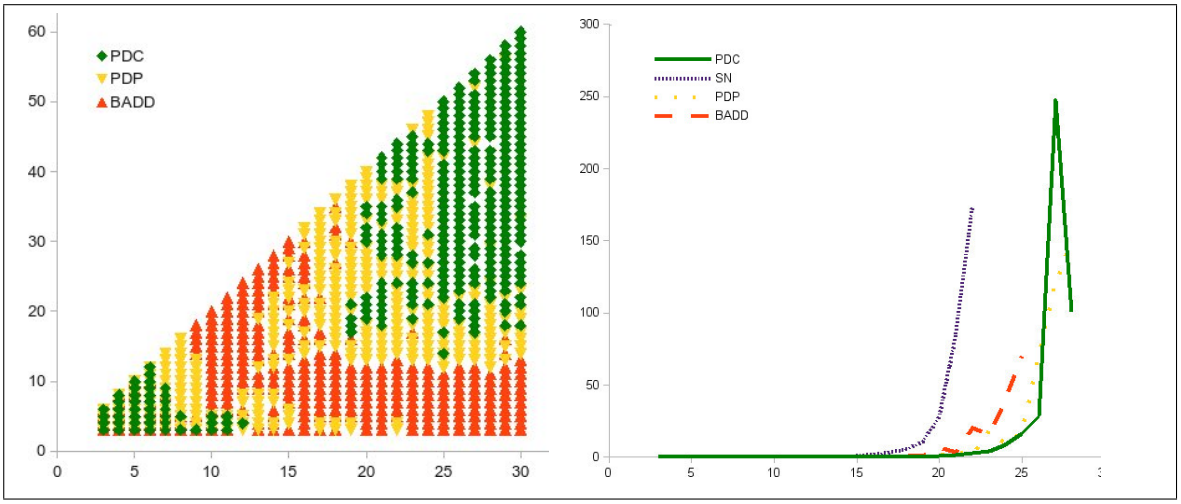

Figure 10.1: The left hand figure plots the best solver for pairs n and L ($n \in \{3, \cdots, 30\}, L \in \{3, \cdots, 2n\}$). The right hand figure shows the average solving time, in seconds, of the engines which solved all the 100 instances in 10 minutes timeout, for $n = L \in \{3, \cdots, n\}$.

2. Sorting network fails to solve all the instances where $n \geq 20$.
3. npSolver also fails to solve all the instances where $n \geq 21$. Moreover, npSolver was not the winner (the best performing solver) in any of the experiments.
4. BADD solves all the instances when $n = L = 24$ in a reasonable time, but it suddenly fails when the values of n and L go above 24.
5. For large enough n ($n > 15$) BADD is the winner only when L is small.
6. For large enough n ($n > 15$) either PDC or PDP is the best performing solver.

The above observations suggest that there are many hard instances for which our encoding outperforms both SN and BADD encodings. As we discussed in the beginning of this section, if L is small there is a polynomial time algorithm for the Number Partitioning problem. Item 5 of the list above suggests that BADD encoding is suitable for small L, which are easy to solve instances. On the other hand, the last observation suggests that our proposed encoding is suitable to attack the hard instances.

We compared our proposed encodings with two of the best available open source PB-solvers. The results of the experiments suggest that both MiniSAT+ and npSolver fail to solve many instances of NPP, while both performed well on the instances from PB-competitions. We should mention that there is a huge difference between the instances used in PB-competition and here. In most cases, an instance used in PB-competition is made of many PB-constraints where each PB-constraint is easy to solve individually, and finding a solution to all the PB-constraints is the main challenge. In our experiments, there is a single hard-to-solve PB-constraint.

Chapter 11

Conclusion and Future Work

In this thesis, we introduced our grounding-based solver, Enfragmo. Enfragmo receives a problem specification and a problem instance as its input and returns a solution for the problem, if any exists. The problem specifications are described in a high-level input language (Chapter 3) which is a fragment of guarded first-order logic expanded with inductive definitions, functions, aggregates and arithmetical operators. In Chapter 3, we also proposed some advice which can be used to develop better performing specifications. The high-level language accepted by Enfragmo can express problems in the complexity class NP, and any problem expressible in this language belongs to the complexity class NP, Chapter 4.

Since every problem expressible in the input language of Enfragmo is an NP-problem, any instance of such problems can be reduced to an instance of an NP-complete problem such as satisfiability. Enfragmo, during its grounding process, generates an equivalent variable-free first-order formula for any given problem specification and problem instance. The details of the grounding process were discussed in Chapters 5 and 6. To handle the grounding, Enfragmo has been equipped with several new data structures for representing the tables, and we also introduced new notions of answers to terms. Using the techniques explained in Chapters 5 and 6, Enfragmo generates a ground formula in polynomial time.

After the grounding phase, Enfragmo enters the MakeCNF phase in which an equivalent CNF formula is created for the ground formula. We describe a novel approach for generating an equivalent CNF formula from the ground formula resulting from the grounding phase. In this thesis, we also explained two new approaches to translating a cardinality constraints to CNF and proposed a family of translations for transforming PB-constraints into CNF.

11.1 Future Work

In Section 4.3, we described the syntax and semantics of a logic and mentioned that although the syntax of that logic and the input language of Enfragmo are the same, their semantics are slightly different. To extend the work described in Chapter 4, one could find a logic which has exactly the same semantics as the input language of Enfragmo.

We proposed several approaches to represent extended relations in Chapter 5, and we showed that they perform differently on different specifications. In the current implementation of Enfragmo,

users can choose the kind of tables to be used by grounder. To extend the work described in this chapter, one could develop methods/heuristics which automatically select the best representation for the tables, given a problem specification and problem instance.

Chapter 6 describes how Enfragmo handles the grounding of the specifications involving complex terms. The author proposed the binary answer to terms to provide a polynomial time grounding. As mentioned in the chapter, the author proposed another notion of answer to terms, which was not discussed in this thesis. The main idea in the other approach is to look at the value of the terms modulo prime numbers. One natural way to extend the work described in this chapter is to implement this representation for answer to term, and compare the performance of these two notions of answers to terms.

Chapter 8 describes how Enfragmo deals with generating CNF from a variable-free first-order formula. We observed that enabling the memorization feature accelerates the CNF generation method for certain problems, and we proposed a heuristic which can be used to automate the decision of enabling this option. To extend the work described in this chapter, one could implement and tune that heuristic.

Chapter 9 proposes two new encodings for translating cardinality constraints to CNF. We showed that our DP-based encoding almost always outperforms BDD-based encoding and our DC-based encodings almost always outperforms Sorting Network-based encoding. We showed that there are problems for which DP-based encoding performs better than DC-based encoding and vice versa. One could develop heuristics to select the best encoding to translate cardinality constraints to CNF.

Chapter 10 proposed a family of translations for transforming PB-Encodings to CNF. As we expressed in the chapter, the bounds in Proposition 27 are not tight and could be improved. The other option to extend the work described in this chapter is to develop methods to select the best PBMod-encoding automatically.

Bibliography

[1] The Asparagus library of examples for ASP programs, http://asparagus.cs.uni-potsdam.de/. 17, 25

[2] http://www2.cs.sfu.ca/~aaa78/personal/Thesis/Experiments/Experiments.html. 25

[3] A. Aavani, D. Mitchell, and E. Ternovska. New encoding for translating Pseudo-Boolean constraints into SAT. In *Proceedings of the ninth Symposium on Abstraction, Reformulation, and Approximation (SARA)*, 2013. 134

[4] A. Aavani, D. Mitchell, and E. Ternovska. Problem Solving with the Enfragmo System. In *ICLP*, 2013. 14

[5] A. Aavani, S. Tasharrofi, G. Ünel, E. Ternovska, and D. Mitchell. Speed-up techniques for negation in grounding. *Logic for Programming, Artificial Intelligence, and Reasoning*, 2010. 41, 95

[6] A. Aavani, N. Wu, D. Mitchell, and E. Ternovska. Grounding count aggregates. *Logic for Programming Artificial Intelligence and Reasoning*, 2010. 17, 122, 148

[7] A. Aavani, N. Wu, E. Ternovska, and D. Mitchell. Grounding formulas with complex terms. *Canadian AI*, 2011. 70, 95

[8] A. Aavani, X. Wu, S. Tasharrofi, E. Ternovska, and D. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In *LPAR-2012*, volume 7180 of *LNCS*, pages 15–22. Springer, 2012. 12

[9] N. Alon and R. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, 7(1):1–22, 1987. 147

[10] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: a backtrack-search Pseudo-Boolean solver and optimizer. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability*, pages 346–353. Citeseer, 2002. 132

[11] M. Alviano, G. Greco, and N. Leone. Dynamic magic sets for programs with monotone recursive aggregates. *Logic Programming and Nonmonotonic Reasoning*, pages 148–160, 2011. 102

[12] O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of Pseudo-Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006. 146

[13] O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of Pseudo-Boolean constraints into CNF. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 181–194, 2009. 123, 136, 146, 147, 152

[14] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985. 102

[15] C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge Univ Pr, 2003. 100

[16] K. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314. ACM, 1968. 125

[17] A. Biere. *Handbook of Satisfiability*, volume 185. Ios PressInc, 2009. 19

[18] A. Blass and Y. Gurevich. On the unique satisfiability problem. *Information and Control*, 55(1):80–88, 1982. 149

[19] C. Borgs, J. Chayes, and B. Pittel. Phase transition and finite-size scaling for the integer partitioning problem. *Random Structures & Algorithms*, 19(3-4):247–288, 2001. 154

[20] M. Cadoli, T. Mancini, and F. Patrizi. SAT as an effective solving technology for constraint problems. *Foundations of Intelligent Systems*, pages 540–549, 2006. 1

[21] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. An ASP system with functions, lists, and sets. *Logic Programming and Nonmonotonic Reasoning*, pages 483–489, 2009. 100, 101

[22] F. Calimeri, S. Cozza, G. Ianni, and N. Leone. Computable functions in ASP: Theory and implementation. *Logic Programming*, pages 407–424, 2009. 101

[23] E. Codd. A relational model of data for large shared data banks. In *Pioneers and Their Contributions to Software Engineering*, pages 61–98. Springer, 2001. 41

[24] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971. 103

[25] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT press, 2001. 60, 61

[26] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*. Citeseer, 1992. 24

[27] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning-International Conference*, pages 148–159, 1996. 24

[28] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)*, 33(3):374–425, 2001. 101

[29] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N Leone, S Perri, and G Pfeifer. System description: DLV. In *LPNMR*, pages 424–428, 2001. 25

[30] M. Denecker and E. Ternovska. A logic of Non-Monotone Inductive Definitions. *Transactions on Computational Logic*, 9(2):1–51, 2008. 11

[31] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczynski. The second answer set programming competition. *LPNMR*, pages 637–654, 2009. 25

[32] C. Ding, D. Pei, and A. Salomaa. *Chinese remainder theorem: applications in computing, coding, cryptography*. World Scientific Publishing Co., Inc. River Edge, NJ, USA, 1996. 151

[33] N. Eén. *SAT Based Model Checking.* PhD thesis, Department of Computing Science, Chalmers University of Technology and Goteborg University, 2005. 106, 107, 108, 115, 116, 117, 125

[34] N. Eén and N. Sorensson. Translating Pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(3-4):1–25, 2006. 142, 148, 152, 153

[35] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems (TODS)*, 22(3):364–418, 1997. 102

[36] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation, SIAM-AMC proceedings*, 7:43–73, 1974. 95

[37] B. Farhi and D. Kane. New results on the least common multiple of consecutive integers. In *Proc. Amer. Math. Soc*, volume 137, pages 1933–1939, 2009. 138

[38] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. In *Proceedings of SymCon*, volume 1. Citeseer, 2001. 24

[39] A. Frisch, M. Grum, C. Jefferson, B.M. Hernández, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proc., Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, 2007. 1

[40] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun. 24(2)*, pages 105–124, 2011. 25

[41] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. *Logic Programming and Nonmonotonic Reasoning*, pages 509–514, 2009. 1

[42] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. *Logic Programming and Nonmonotonic Reasoning*, pages 266–271, 2007. 101

[43] I. Gent and T. Walsh. Analysis of heuristics for number partitioning. In *Computational Intelligence*, volume 14, pages 430–451. Wiley Online Library, 1998. 153

[44] E. Graedel and Y. Gurevich. Metafinite model theory. *Inf. Comput.*, 140(1):26–81, 1998. 7

[45] G. Hardy, E. Wright, D. Heath-Brown, and J. Silverman. *An introduction to the theory of numbers*, volume 6. Clarendon press Oxford, 1979. 138

[46] D. Jackson. *Software Abstractions: logic, language and analysis*. The MIT Press, 2006. 95

[47] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733. IEEE, 2000. 95

[48] R. Kulkarni and P. Bhave. Integer programming formulations of vehicle routing problems. *European Journal of Operational Research*, 20(1):58–67, 1985. 133

[49] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):562, 2006. 100

[50] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004. 7

[51] Y. Lierler and V. Lifschitz. One more decidable class of finitely ground programs. *Logic Programming*, pages 489–493, 2009. 101

[52] Y. Lierler and V. Lifschitz. Termination of grounding is not preserved by strongly equivalent transformations. *Logic Programming and Nonmonotonic Reasoning*, pages 205–210, 2011. 101

[53] N. Manthey and P. Steinke. npSolver–A SAT based solver for optimization problems. *Pragmatics of SAT (POS'12)*, 2012. 153

[54] D. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. AAAI*, 2005. 6, 12, 27, 91

[55] D. Mitchell and E. Ternovska. Expressive power and abstraction in ESSENCE. *Constraints*, 13(3):343–384, 2008. 9

[56] R. Mohebali. A method for solving NP search based on model expansion and grounding. Master's thesis, Simon Fraser University, 2006. 40, 41, 43, 46, 47, 49, 54, 60, 66, 70, 94

[57] C. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 255–260. ACM, 1982. 149

[58] M. Patterson, Y. Liu, E. Ternovska, and A. Gupta. Grounding for model expansion in k-Guarded formulas with inductive definitions. In *Proc. IJCAI'07*, pages 161–166, 2007. 40, 41, 42, 43, 44, 68, 70, 92, 93, 94

[59] Nikolay Pelov and Eugenia Ternovska. Reducing inductive definitions to propositional satisfiability. In *Logic Programming*, pages 221–234. Springer, 2005. 16

[60] T Sálat and S Znám. On the sums of prime powers. *Acta Fac. Rer. Nat. Univ. Com. Math*, 21:21–25, 1968. 145

[61] H. Sheini and K. Sakallah. Pueblo: A hybrid Pseudo-Boolean SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–96, 2006. 132

[62] P. Simons, I. Niemelá, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. 1

[63] T. Syrjänen. Omega-restricted logic programs. *Logic Programming and Nonmotonic Reasoning*, pages 267–280, 2001. 101

[64] S. Tasharrofi and E. Ternovska. Capturing NP for search problems with built-in arithmetic. *LPAR-17. LNCS*, 6397, 2009. 36, 70

[65] E. Ternovska and D. Mitchell. Declarative programming of search problems with built-in arithmetic. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 942–947, 2009. 3, 5, 6, 7, 8, 9, 10, 11, 15, 27, 30, 69

[66] E. Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008. 24, 95

[67] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic*, 2(115-125):10–13, 1968. 3, 104, 105

[68] http://minisat.se/, 2008. 153

[69] P. Vaezipoor, D. Mitchell, and N. Mariën. Lifted unit propagation for effective grounding. *19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, 2011. 24

[70] A. Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008. 19

[71] F. Van Harmelen, V. Lifschitz, and B. Porter. *Handbook of knowledge representation*. Elsevier, 2008. 1

[72] J. Wittocx, M. Mariën, and M. Denecker. Grounding with bounds. In *Proc. AAAI'08*, pages 572–577, 2008. 1, 97, 98, 99

[73] J. Wittocx, M. Marién, and M. Denecker. The IDP system: A model expansion system for an extension of classical logic. In *Proceedings of the 2nd Workshop on Logic and Search*, pages 153–165, 2008. 16, 25

[74] J. Wittocx, M. Mariën, and M. Denecker. Grounding FO and FO(ID) with bounds. *J. Artif. Int. Res.*, 38(1):223–269, May 2010. 18

[75] N. Wu and L. Swanson. *The Enfragmo System*, 2011. http://www.cs.sfu.ca/~ter/eternovska/Software_files/Enfragmo-man.pdf. 14

[76] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008. 103

Appendix A

The Input Language of Kodkod

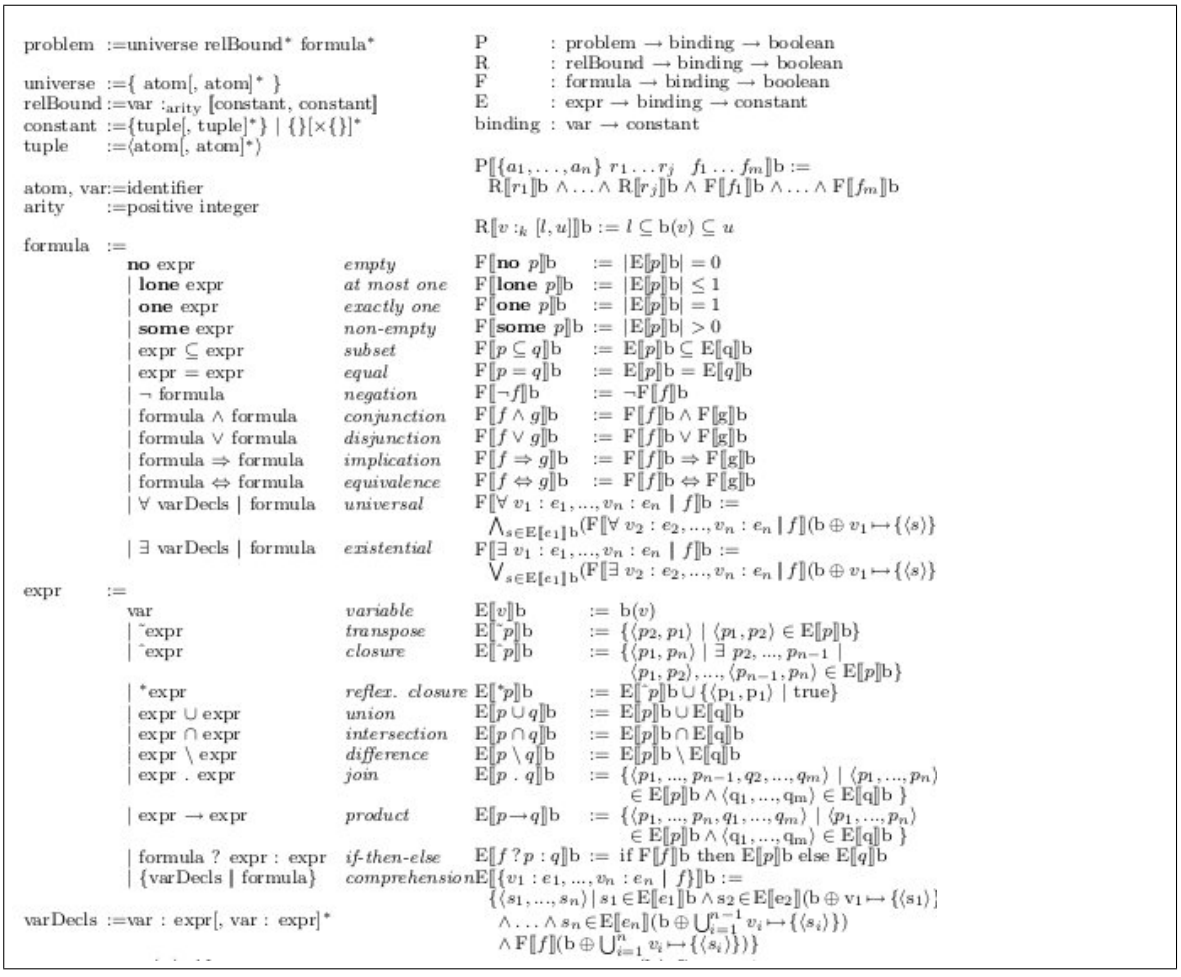

Figure A.1: Kodkod Input Language

Appendix B

Translation Rules for Kodkod

$\mathrm{T_P}$: problem $\to$ bool
$\mathrm{T_R}$: relBound $\to$ universe $\to$ matrix
$\mathrm{T_F}$: formula $\to$ env $\to$ bool
$\mathrm{T_E}$: expr $\to$ env $\to$ matrix
env : var $\to$ matrix

bool := 0 | 1 | boolVar | $\neg$ bool | bool $\wedge$ bool | bool $\vee$ bool | bool ? bool : bool
boolVar := identifier
idx := $\langle$int[, int]$^*\rangle$

$\mathcal{V}$: var $\to$ $\langle$atom[, atom]$^*\rangle$ $\to$ boolVar — *boolean variable for a given tuple in a relation*
$(\!|\,|\!)$: matrix $\to$ {idx[, idx]*} — *set of all indices in a matrix*
$[\![\,]\!]$: matrix $\to$ int$^{\text{int}}$ — *size of a matrix, (size of a dimension)$^{number\ of\ dimensions}$*
[ ] : matrix $\to$ idx $\to$ bool — *matrix value at a given index*

$\mathcal{M}$: int$^{\text{int}}$ $\to$ (idx $\to$ bool) $\to$ matrix
$\mathcal{M}(s^d, f) := \text{new } m \in \text{matrix where } [\![m]\!] = s^d \wedge \forall\, \vec{x} \in \{0,...,s-1\}^d, m[\vec{x}] = f(\vec{x})$

$\mathcal{M}$: int$^{\text{int}}$ $\to$ idx $\to$ matrix
$\mathcal{M}(s^d, \vec{x}) := \mathcal{M}(s^d, \lambda \vec{y}.\ \text{if } \vec{y} = \vec{x} \text{ then 1 else 0})$

$\mathrm{T_P}[\{a_1,\ldots,a_n\}\ \ v_1 :_{k_1} [l_1,u_1] \ldots v_j :_{k_j} [l_j,u_j]\ \ f_1 \ldots f_m] := \mathrm{T_F}[\bigwedge_{i=1}^{m} f_i](\cup_{i=1}^{j} v_i \mapsto \mathrm{T_R}[v_i :_{k_i} [l_i,u_i], \{a_1,\ldots,a_n\}])$

$\mathrm{T_R}[v :_k [l,u], \{a_1,\ldots,a_n\}] := \mathcal{M}(n^k, \lambda\langle i_1,\ldots i_k\rangle.\ \text{if } \langle a_{i_1},\ldots,a_{i_k}\rangle \in l \text{ then 1}$
$\text{else if } \langle a_{i_1},\ldots,a_{i_k}\rangle \in u \setminus l \text{ then } \mathcal{V}(v, \langle a_{i_1},\ldots,a_{i_k}\rangle)$
$\text{else 0})$

$\mathrm{T_F}[\textbf{no } p]\mathrm{e} := \neg \mathrm{T_F}[\textbf{some } p]\mathrm{e}$
$\mathrm{T_F}[\textbf{lone } p]\mathrm{e} := \mathrm{T_F}[\textbf{no } p]\mathrm{e} \vee \mathrm{T_F}[\textbf{one } p]\mathrm{e}$
$\mathrm{T_F}[\textbf{one } p]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[p]\mathrm{e} \text{ in } \bigvee_{\vec{x} \in (\!|m|\!)} m[\vec{x}] \wedge (\bigwedge_{\vec{y} \in (\!|m|\!) \setminus \{\vec{x}\}} \neg m[\vec{y}])$
$\mathrm{T_F}[\textbf{some } p]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[p]\mathrm{e} \text{ in } \bigvee_{\vec{x} \in (\!|m|\!)} m[\vec{x}]$
$\mathrm{T_F}[p \subseteq q]\mathrm{e} := \text{let } m \leftarrow (\neg \mathrm{T_E}[p]\mathrm{e} \vee \mathrm{T_E}[q]\mathrm{e}) \text{ in } \bigwedge_{\vec{x} \in (\!|m|\!)} m[\vec{x}]$
$\mathrm{T_F}[p = q]\mathrm{e} := \mathrm{T_F}[p \subseteq q]\mathrm{e} \wedge \mathrm{T_F}[q \subseteq p]\mathrm{e}$
$\mathrm{T_F}[\textbf{not } f]\mathrm{e} := \neg \mathrm{T_F}[f]\mathrm{e}$
$\mathrm{T_F}[f \wedge g]\mathrm{e} := \mathrm{T_F}[f]\mathrm{e} \wedge \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \vee g]\mathrm{e} := \mathrm{T_F}[f]\mathrm{e} \vee \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \Rightarrow g]\mathrm{e} := \neg \mathrm{T_F}[f]\mathrm{e} \vee \mathrm{T_F}[g]\mathrm{e}$
$\mathrm{T_F}[f \Leftrightarrow g]\mathrm{e} := (\mathrm{T_F}[f]\mathrm{e} \wedge \mathrm{T_F}[g]\mathrm{e}) \vee (\neg \mathrm{T_F}[f]\mathrm{e} \wedge \neg \mathrm{T_F}[g]\mathrm{e})$

$\mathrm{T_F}[\forall\, v_1 : e_1, ..., v_n : e_n \mid f]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[e_1]\mathrm{e} \text{ in } \bigwedge_{\vec{x} \in (\!|m|\!)} (\neg m[\vec{x}] \vee \mathrm{T_F}[\forall\, v_2 : e_2, ..., v_n : e_n \mid f](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}([\![m]\!], \vec{x})))$
$\mathrm{T_F}[\exists\, v_1 : e_1, ..., v_n : e_n \mid f]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[e_1]\mathrm{e} \text{ in } \bigvee_{\vec{x} \in (\!|m|\!)} (m[\vec{x}] \wedge \mathrm{T_F}[\forall\, v_2 : e_2, ..., v_n : e_n \mid f](\mathrm{e} \oplus v_1 \mapsto \mathcal{M}([\![m]\!], \vec{x})))$

$\mathrm{T_E}[v]\mathrm{e} := \mathrm{e}(v)$
$\mathrm{T_E}[\tilde{}\,p]\mathrm{e} := (\mathrm{T_E}[p]\mathrm{e})^{\mathsf{T}}$
$\mathrm{T_E}[\hat{}\,p]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[p]\mathrm{e}, s^d \leftarrow [\![m]\!], \mathrm{sq} \leftarrow (\lambda x.i.\ \text{if } i = s \text{ then } x \text{ else let } y \leftarrow \mathrm{sq}(x, i*2) \text{ in } y \vee y \cdot y) \text{ in } \mathrm{sq}(m, 1)$
$\mathrm{T_E}[{}^*p]\mathrm{e} := \text{let } m \leftarrow \mathrm{T_E}[\hat{}\,p]\mathrm{e}, s^d \leftarrow [\![m]\!] \text{ in } m \vee \mathcal{M}(s^d, \lambda\langle i_1,\ldots,i_d\rangle.\ \text{if } i_1 = i_2 \wedge \ldots \wedge i_1 = i_k \text{ then 1 else 0})$
$\mathrm{T_E}[p \cup q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \vee \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \cap q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \wedge \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \setminus q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \wedge \neg \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \,.\, q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \cdot \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[p \to q]\mathrm{e} := \mathrm{T_E}[p]\mathrm{e} \times \mathrm{T_E}[q]\mathrm{e}$
$\mathrm{T_E}[f\ ?\ p : q]\mathrm{e} := \text{let } m_p \leftarrow \mathrm{T_E}[p]\mathrm{e}, m_p \leftarrow \mathrm{T_E}[q]\mathrm{e} \text{ in } \mathcal{M}([\![m_p]\!], \lambda\vec{x}.\ \mathrm{T_F}[f]\mathrm{e}\ ?\ m_p[\vec{x}]\ :\ m_q[\vec{x}])$

Appendix C

Computing Bounds in IDP

Algorithm 11 The following program is used by IDP to compute $ctb(\psi)$ and $cfb(\psi)$ of all subformulas ψ.

Input: A theory T over Σ, a symbolic $\Sigma-$structure Φ and a constant $C \in N$.

Q:= 0
for $P \in \Sigma^{tf}_{pred}$ **do**
  if $P^{\Phi^{tf}}$ is not of the form $\{\bar{x}|\bot\}$ **then**
    for all $(\forall \bar{x}(\psi \Rightarrow L[\bar{x}])) \in INF(T)$ such that P occurs in ψ^{ct} **do**
      $Q.push(\forall \bar{x}(\psi \Rightarrow L[\bar{x}]))$
n:= 0
while $Q \neq \emptyset$ and $n < C$ **do**
  $\forall \bar{x}\ (\psi \Rightarrow L[\bar{x}] := Q.pop())$
  if $L[\bar{x}]$ is a positive literal $P(\bar{x})$ **then**
    $q := P^{\Phi^{ct}} \cup \{\bar{x}|\Phi^{ct}(\psi^{ct})\}$
    $q :=$ simplify (q)
    if (not equivalent $(q, P^{\Phi^{ct}})$) and acceptable $(q, P^{\Phi^{ct}})$ **then**
      $n := n + 1$
      $P^{\Phi^{ct}} := q$
      for all $(\forall \bar{y}(\chi \Rightarrow L'[\bar{y}])) \in (INF(T) \backslash Q)$ such that P occurs positively in χ^{ct} **do**
        $Q.push(\forall \bar{y}(\chi \Rightarrow L'[\bar{y}]))$
  elseSimilar code when $L[\bar{x}]$ is a negative literal.

Appendix D

Grounding Using Bounds in IDP

Algorithm 12 Grounding with bounds

Input: T, σ, I_σ, C

Output: A grounding T_g for T with respect to I_σ

if $C^{cfb}(\phi) = \top$ for some axiom ϕ of T **then return** $\bot$

$T_g := \emptyset$

Ground all sentences of T

for every sentence ϕ of T **do**

  if $C^{ctb}(\phi) \neq \top$ **then**

    Add GroundConj (ϕ) to T_g

Add the grounding of C_A

for every atomic subformula $\phi[\bar{x}]$ of T **do**

  for every $\bar{d}$ such that $I_\sigma[\bar{x}/\bar{d}] \models C^{ctb}(\phi)$ **do**

    Add $\phi[\bar{x}/\bar{d}]$ to T_g

  for every $\bar{d}$ such that $I_\sigma[\bar{x}/\bar{d}] \models C^{cfb}(\phi)$ **do**

    Add $\neg\phi[\bar{x}/\bar{d}]$ to T_g

---

function G(r)oundConj ($\phi[\bar{x}]$)
  $C := \emptyset$
  Switch$\phi[\bar{x}]$
  Case$\phi = \forall y \Psi[\bar{x}, y]$ **return** GroundConj ($\Psi[\bar{x}, y]$)
  Case$\phi = \bigwedge_i \Psi_i$
  $C := \bigvee_i GroundConj(\Psi_i)$
  Other
  for all $\bar{d}$ such that $I_\sigma \not\models C^{ctb}(\phi)[\bar{x}/\bar{d}]$ **do**
    if $I_\sigma \models C^{cfb}(\phi)[\bar{x}/\bar{d}]$ **then return** $\bot$
    else
      if ϕ is a literal **then**
        Add $\phi(\bar{x}/\bar{d})$ to C
      else
        if ϕ is a disjunctive formula **then**
          Add GroundDisj $\phi(\bar{x}/\bar{d})$ to C
        else
          if ϕ is an aggregate expression **then** Add GroundAgg $\phi(\bar{x}/\bar{d})$ to C
  return $\bigwedge C$

---

function G(r)oundDisj ($\phi[\bar{x}]$)
  $D := \emptyset$
  Switch$\phi[\bar{x}]$
  Case$\phi = \exists y \Psi[\bar{x}, y]$ **return** GroundDisj ($\Psi[\bar{x}, y]$)
  Case$\phi = \bigvee_i \Psi_i$
  $C := \bigvee_i GroundDisj(\Psi_i)$
  Other
  for all $\bar{d}$ such that $I_\sigma \not\models C^{cfb}(\phi)[\bar{x}/\bar{d}]$ **do**
    if $I_\sigma \models C^{ctb}(\phi)[\bar{x}/\bar{d}]$ **then return** $\top$
    else
      if ϕ is a literal **then**
        Add $\phi(\bar{x}/\bar{d})$ to D
      if ϕ is a conjunctive formula **then**
        Add GroundDisj $\phi(\bar{x}/\bar{d})$ to D
      if ϕ is an aggregate expression **then**
        Add GroundAgg $\phi(\bar{x}/\bar{d})$ to D
  return $\bigvee D$

Appendix E

The Input Language of Enfragmo

E.1 Problem Specification Grammar

```
<theory_file> ::= <given_part> <find_part>
                                        <phase_part> <print_part>

<given_part> ::= GIVEN : <types_decl> ; <funcs_decl> ;
           | GIVEN : <types_decl> ; <preds_decl> ;
           | GIVEN : <types_decl> ; <preds_decl> ;
                                                    <funcs_decl> ;

<types_decl> ::= TYPES : <identifier_list> ;
                                     INTTYPES : <identifier_list>
           | TYPES : <identifier_list>
               | INTTYPES : <identifier_list>

<identifier_list> ::= | <identifier_list> <identifier>

<preds_decl> ::= PREDICATES : <preds_list>

<a_pred_DCL> ::= <identifier> (<IdentifierListSeparatedByComma>)

<preds_list> ::= | <a_pred_DCL> <preds_list>

<IdentifierListSeparatedByComma> ::= | <identifier>
               | <IdentifierListSeparatedByComma> , <identifier>

<funcs_decl> ::= FUNCTIONS : <funcs_list>

<funcs_list> ::= | <func_DCL> <funcs_list>

<func_DCL> ::=   <identifier> ( ) : <identifier>
 | <identifier> (<IdentifierListSeparatedByComma>): <identifier>
```

```
<find_part> ::= FIND : <identifier_list> ;

<phase_part> ::= <a_phase> | <a_phase>  <phase_part>

<a_phase> ::= PHASE : <fixpoint_part> <satisfying_part>

<fixpoint_part> ::= | FIXPOINT ( <identifier_list> ) :
                                                <induction_part> ;

<induction_part> ::= <an_induction>
                               | <induction_part> <an_induction>

<an_induction> ::= <an_inflation> | <a_definition>

<an_inflation> ::= INFLATE <inflate_description>

<an_inflate_description> ::= { <var_DCL> : <identifier>
         ( <IdentifierListSeparatedByComma> ) <=> <FO_formula> }

<inflate_description> ::= <an_inflate_description>
                | <inflate_description> <an_inflate_description>

<a_definition> ::= DEFINE { <induction_description> }

<induction_description> ::= <an_induction_description>
            | <induction_description> <an_induction_description>

<an_induction_description> ::= <var_DCL> : <identifier>
            ( <IdentifierListSeparatedByComma> ) <- <FO_formula>

<satisfying_part> ::= | SATISFYING : <satisfying_rules>

<satisfying_rules> ::= <FO_formula> ;
                             | <satisfying_rules> <FO_formula> ;

<FO_formula> ::= ( <FO_formula> ) | <unitary_formula>
           | <FO_formula> <connective> <unitary_formula>

<unitary_formula> ::= ( <FO_formula> )
         | <quantifier> <var_DCL> : <FO_formula>
         | <quantifier> <a_var_DCL> <ord_operator>
                                  <term_nodes>: <unitary_formula>
         | <unitary_formula> <binary_operator> <unitary_formula>
         | ~ <unitary_formula>
         | <atomic_formula>

<atomic_formula> ::= <relation_formula>
         | SUCC [ <identifier> ] ( <term_nodes> , <term_nodes> )
         | <ord_relation> | TRUE | FALSE
```

```
<relation_formula> ::= <identifier> ( <args> )

<ord_relation> ::= <term_nodes> <ord_operator> <term_nodes>

<min_func> ::= MIN [ <identifier> ]

<max_func> ::= MAX [ <identifier> ]

<size_func> ::= SIZE [ <identifier> ]

<abs_func> ::= ABS ( <term_nodes> )

<func_ref> ::= <identifier> ( <args> ) | <identifier> ( )

<var_DCL> ::= | <a_var_DCL> | <var_DCL> <a_var_DCL>

<a_var_DCL> ::= <identifier> : <identifier>

<args> ::= | <term_nodes> | <args> , <term_nodes>

<term_nodes> ::= <a_term_node> | ( <term_nodes> )
               | <term_nodes> + <term_nodes>
               | <term_nodes> * <term_nodes>
               | <term_nodes> - <term_nodes>

<a_term_node> ::= <var_ref> | <min_func> | <max_func>
                | <abs_func> | <func_ref> | <size_func>
                | <aggregate> | <int_term_node>

<aggregate> ::= COUNT { <var_DCL> ; <FO_formula> }
| MIN { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
| MAX { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
| SUM { <var_DCL> ; <term_nodes> ; <FO_formula> }

<var_ref> ::= <identifier>

<int_term_node> ::= <int_number> | INTEGER { <term_nodes> }

<arit_operator> ::= + | * | -

<quant_part> ::= <quantifier> <var_DCL>;

<quantifier> ::= ? | !

<binary_operator> ::= & | '|' (or)

<ord_operator> ::= < | <= | > | >= | =
```

```
<connective> ::= & | '|' (or) | => | <=>

<print_part> ::= | PRINT : <predicates>

<predicates> ::= | <predicates> <identifier>

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+
```

E.2 Instance Specification Grammar

```
<instance_file> ::= <type_parts> <pred_parts> <func_parts>

<type_parts> ::= <a_type_part> | <type_parts> <a_type_part>

<a_type_part> ::= TYPE <identifier> <range>

<range> ::= [ continous_values ]

<continous_values> ::= <integer>..<integer>

<discrete_values> ::= <a_discrete_value>
                    | <discrete_values>, <a_discrete_value>

<a_discrete_value> ::= <integer> | <string>

<pred_parts> ::= | <pred_parts> <a_predicate_part>

<a_predicate_part> ::= PREDICATE <identifier> <predicate_values>

<predicate_values> ::= | <a_predicate_value>
                     | <predicate_values> <a_predicate_value>

<a_predicate_value> ::= (<discrete_values>)

<func_parts> ::= | <func_parts> <a_function_part>

<a_function_part> ::= FUNCTION <identifier> <function_values>

<function_values> ::= <a_function_value>
                    | <function_values> <a_function_value>

<a_function_value> ::= (<func_args>:<func_return_value>)

<func_args> ::= | <discrete_values>

<func_return_value> ::= <integer> | <string>
```

```
<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+

<string> ::= '[0-9a-zA-Z_]+'

<integer> ::= [0-9]+
```

Appendix F

Linear Sorting Algorithm for Tables

Input: Set of variables X, Table T representing an X-extended relation, Variable ordering O
Output: The set of rows in $TargetTable$ is the same as the set of rows in T, sorted based on variable ordering O

$W := |X|$
$H := |S|$
SourceTable= T
TargetTable= T
for $c \leftarrow W - 1, 0$ **do**
  Min= The minimum value in column c of $SourceTable$;
  Max= The maximum value in column c of $SourceTable$;
  Let $Count$ be an array with $Max + 1$ elements.
  for $r \leftarrow 0, H - 1$ **do**
    Set v to be the element in row r column c of table $SourceTable$.
    Increase $Count[v]$ by one.
  Let $Head$ be an array with $Max + 1$ elements.
  $Head[0] = 0$
  for $i \leftarrow 0, Max - 1$ **do**
    $Head[i + 1] = Head[i] + Count[i]$
  for $r \leftarrow 0, H - 1$ **do**
    $Index = SourceTable[r][c]$
    $TargetTable[Head[Index]] = SourceTable[r]$
    Increase $Head[Index]$ by one
  Swap SourceTable and TargetTable
return TargetTable
