

LOGIC PROGRAMS AND CARDINALITY CONSTRAINTS

Theory and Practice

Tommi Syrjänen



TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

LOGIC PROGRAMS AND CARDINALITY CONSTRAINTS

Theory and Practice

Tommi Syrjänen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 20 of March, 2009, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: <http://ics.tkk.fi>
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi

© Tommi Syrjänen

ISBN 978-951-22-9762-7 (Print)
ISBN 978-951-22-9763-4 (Online)
ISSN 1797-5050 (Print)
ISSN 1797-5069 (Online)
URL: <http://lib.tkk.fi/Diss/2009/isbn9789512297634/>

TKK ICS
Espoo 2009

ABSTRACT: Answer set programming (ASP) is a method for solving hard problems using computational logic. We describe a problem as a set of formulas of a declarative logical language in such way that the solutions correspond to the models (answer sets) of the set and then use a general-purpose inference engine to find the answer sets.

In this work we define an ASP language, *cardinality constraint programs* (CCP). The language extends normal logic programs by adding cardinality and conditional literals as well as choice rules. These extensions allow us to represent many if not most **NP**-complete problems in a concise and intuitive way. The language is defined in two phases where we first introduce a simple basic language and then define the constructs of the full language in terms of translations to the basic language.

The language has a declarative formal semantics that is based on the stable model semantics of normal logic programs. The semantics of a program with variables is defined via its ground instantiation. In addition of using the Herbrand instantiation a program can be instantiated with respect to some other universe, which makes it possible to have a direct support for interpreted functions in the semantics.

The semantics is undecidable in the general case. We identify a syntactic subclass of CCPs, namely *omega-restricted programs*, that are decidable even when function symbols are allowed. The stable models of such programs are created by a finite relevant instantiation that we can always compute. We analyze the computational complexity of omega-restricted programs and show that deciding whether a program has a stable model is 2-**NEXP**-complete. We identify further subclasses of programs that are **NP**- and **NEXP**-complete in the same sense. We also present an algorithm for instantiating omega-restricted programs.

We discuss programming methodology and show how we can create uniform CCP encodings for different problems using the generate-and-test methodology. We examine how we can combine ASP with traditional programming languages by treating an ASP solver as an oracle.

As an extended case study we examine how we can solve AI planning problems using ASP. We present a systematic translation from an action language planning formalism into CCPs and give encodings for three sample planning domains. We also examine how we can add domain-specific knowledge to the encodings to make them more efficient in practice.

KEYWORDS: Computational Logic, Logic Programming, Answer Set Programming, Stable Model Semantics, Cardinality Constraints, Planning

TIIVISTELMÄ: Vastausjoukko-ohjelmointi (answer set programming, ASP) on menetelmä laskennallisesti vaativien ongelmien ratkaisemiseksi matemaattisella logiikalla. Ongelma esitetään jonkin loogisen kielen lausejoukkona siten, että sen ratkaisut vastaavat lausejoukon malleja. Mallien laskemiseen käytetään yleiskäyttöistä päättelykonetta.

Työssä määritellään ASP-kieli, *kardinaliteettirajoiteohjelmat* (cardinality constraint programs, CCP), joka laajentaa logiikkaohjelmia lisäämällä niihin kardinaliteetti- ja ehdolliset literaalit sekä valintasäännöt. Laajennusten avulla voidaan muodostaa suurelle osalle **NP**-täydellisiä ongelmia tiivis ja intuitiivisesti selkeä esitys.

Kieli määritellään kahdessa vaiheessa. Ensimmäiseksi esitetään yksinkertainen peruskieli ja sen semantiikka. Täyden kielen rakenteet määritellään kääntämällä ne peruskielen rakenteiksi.

Kielen semantiikka perustuu logiikkaohjelmien stabiilien mallien semantiikkaan. Muuttujallisen ohjelman semantiikka määräytyy sen instantiaation kautta. Tavanomaisen Herbrand-instantiaation lisäksi instantiaatio voidaan tehdä jonkin toisen universumin suhteen, jolloin siihen voidaan suoraan upottaa tulkitut funktiosymbolit.

Stabiilien mallien semantiikka CCP-ohjelmille ei ole ratkeava yleisessä tapauksessa. Tämän vuoksi työssä määritellään kielelle ratkeava aliluokka, *omega-rajoitetut ohjelmat*, joiden stabiilien mallien määräämiseen riittää instantiaation äärellinen osajoukko. Työssä esitetään algoritmi, jolla voidaan tuottaa instantiaation oleellinen osa.

Työssä analysoidaan omega-rajoitettujen ohjelmien laskennallista vaativuutta. Yleisessä tapauksessa ohjelmat ovat **2-NEXP**-täydellisiä, mutta niille määritellään myös **NP**- ja **NEXP**-täydelliset alaluokat.

Työssä tarkastellaan yleiskäyttöisten CCP-ohjelmien laatimiseen liittyviä käytännön seikkoja ja esitetään ”arvaa ja tarkista” -periaatteen pohjautuva menetelmä ongelmien esittämiseen. Lisäksi esitetään menetelmä, jolla ASP-ohjelma voidaan yhdistää perinteisellä kielellä kirjoitettuun ohjelmaan käyttämällä ASP-toteutusta oraakkelinä.

Laajana sovellusesimerkinä tarkastellaan ASP-ohjelmien käyttämistä suunnitteluongelmien ratkaisemiseen. Ongelma formalisoidaan suunnittelukuvauskielellä, joka puolestaan käännetään systemaattisesti CCP-ohjelmaksi. Työssä esitetään kolmen suunnitteluongelman logiikkaohjelmakäännökset. Lisäksi tarkastellaan, miten ongelman erityispiirteet voidaan ottaa huomioon ohjelmaa laadittaessa, jotta sen mallit voitaisiin laskea mahdollisimman tehokkaasti.

AVAINSANAT: Laskennallinen logiikka, logiikkaohjelmointi, vastausjoukko-ohjelmointi, stabiilien mallien semantiikka, kardinaliteettirajoitteet, suunnittelu

CONTENTS

1	Introduction	2
1.1	Logic Programming and ASP	3
1.2	Cardinality Constraint Programs	4
1.3	Scientific Contributions	6
1.4	Related Work	7
1.5	Outline of the Work	9
2	Cardinality Constraint Programs	12
2.1	Syntax	12
2.2	Notational Conventions	15
3	The Stable Model Semantics	18
3.1	Normal Logic Programs	18
3.2	Satisfaction and Classical Models	19
3.3	Reducts	19
3.4	The Provability Operator	22
3.5	The Stable Model Semantics of Ground Programs	25
3.6	Some Properties of Cardinality Constraint Programs	27
3.7	Programs with Variables	29
3.8	Interpreted Function Symbols	33
3.9	The Standard Interpretation	39
4	Omega-Restricted Programs	43
4.1	Basic Concepts	44
4.2	Dependency Graphs	45
4.3	Omega-Stratification	47
4.4	Domain Predicates	48
4.5	Omega-Valuation and Restriction	50
4.6	Computing Domain Predicates	51
5	Decidability of Omega-Restricted Programs	55
5.1	Stratum Programs	56
5.2	Uniqueness of Domain Program	60
5.3	Relevant Instantiation	60
5.4	Domain Models	62
5.5	Finiteness of Domain Models	62
5.6	Stable Models of Omega-Restricted Programs	63
5.7	Putting it All Together	64
5.8	Instantiation as a Database Operation	66
6	Computational Complexity	74
6.1	Basics of Computational Complexity	75
6.2	Relationship of MODEL and INSTANTIATION	76
6.3	Turing Machine Translation	77
6.4	The Turing Machine Encoding	79
6.5	Preliminaries for Complexity Proofs	86

6.6	Complexity Results	91
6.7	The Function Version of MODEL	101
7	The Full Language	107
7.1	Language Design	107
7.2	The Full Language Syntax	118
7.3	Transformations	119
7.4	Further Extensions	127
8	Implementation Issues	132
8.1	A Hierarchy of Languages	132
8.2	Overview of the Implementation Architecture	133
8.3	Instantiating Rules	135
8.4	Domain Computation	147
8.5	The <i>smodels</i> rules	155
9	Programming Methodology	157
9.1	Generate and Test Method	157
9.2	Uniform Encodings	158
9.3	Using Answer Set Solvers as Oracles	163
9.4	On Optimization	166
10	Encoding Planning Problems	173
10.1	General Approach	173
10.2	Different Forms of Planning	174
10.3	Formalizing Planning	174
10.4	Translating Action Language \mathcal{B} to ASP	181
10.5	Issues on Parallel Planning	191
10.6	Introducing Variables	197
10.7	Plan Generation	206
10.8	More on Planning Variants	207
10.9	Two Planning Examples	210
11	Conclusions	223
	Bibliography	225
	Index	243

List of Figures

3.1	From the Herbrand universe to an arbitrary universe . . .	33
4.1	The dependency graph of the Hamiltonian cycle program.	46
4.2	A strict ω -stratification of the Hamiltonian cycle program .	48
4.3	An example of SCC graph formation.	52
4.4	An algorithm for creating an ω -stratification	53
4.5	A sample SCC graph and its stratification	53
5.1	A naive algorithm for testing the existence of a stable model of an ω -restricted CCPs.	65
5.2	A naive algorithm for computing the Datalog semantics . .	69
6.1	Complexity classes	76
6.2	A naive algorithm for solving INSTANTIATION	90
6.3	The search tree from Example 6.7.1	103
6.4	A search Tree for Example 6.7.2	106
7.1	The process of computing answer sets	107
7.2	The principle of uniform encodings	109
7.3	A non-modular transformation	109
7.4	A modular transformation	110
8.1	Computing answer sets	132
8.2	The language hierarchy	133
8.3	General level architecture	134
8.4	The instantiation algorithm	134
8.5	Overview of rule instantiation	135
8.6	Instantiating rules	136
8.7	Instantiating local variables	142
8.8	Domain computation	148
8.9	Algorithm for Computing the Domain Model	148
8.10	Visiting the strongly connected components	149
9.1	Example of VERTEX COLORING.	158
9.2	The basic oracle algorithm	163
9.3	The general form for using an oracle	164
9.4	An algorithm for computing functional MAXSAT	164
9.5	Oracle with exclusion sets	165
9.6	A naive way of computing all answer sets	165
9.7	Solving 2-Quantified Boolean Formulas	166
9.8	Sample search trees	167
9.9	A cross-sum/Kakuro puzzle and a solution	168
9.10	Kakuro input encoding	168
9.11	Two choices that fix a unique solution.	171
10.1	Solving planning problems	173
10.2	The LTS corresponding to a two-block BLOCKS WORLD .	178
10.3	The action language translation in a nutshell	182
10.4	The “Sussman Anomaly” BLOCKS WORLD instance	185
10.5	The action description for BLOCKS WORLD	199
10.6	A naive algorithm for plan generation	207
10.7	A vacuum world example	211
10.8	The dynamic laws of VACUUM WORLD	214

10.9 The static laws of VACUUM WORLD	215
10.10A Sokoban instance and an optimal solution	216
10.11An example of Sokoban segments	217

PREFACE

The history of formal logic goes back to the 4th century BC. For the vast majority of that time logicians have made their proofs by hand, first by using Aristotelean syllogisms, and after the advances of the 19th century with truth tables or axiom schemata. These methods are laborious and can be used only for small examples—the truth table of a formula with only six propositional atoms already has 64 rows and each additional atom doubles its size.

Computational logic is a young branch of logic whose ultimate purpose is to solve practical problems with mathematical logic. The formalizations of most problems are far too large to be examined by hand as they often have thousands, hundreds of thousands, or even millions of atoms. We can tackle such formalizations only with the help of a computer.

The workflow for solving problems with computational logic is that we first identify what are the most important properties of our problem domain and express them using some logical language. Then, we use some general purpose solver to compute our answer. From the viewpoint of a common user the solver is a black box: a formalization goes in and the answers come out. The user does not need to know much about the internal details of the solver.

This work examines computational logic in the form of answer set programming (ASP) that is an offshoot of logic programming. We look at the issues that arise in defining, implementing, and using an answer set language.

I'm thankful to my advisor Professor Ilkka Niemelä for the opportunity to work on this interesting topic in the Laboratory for Theoretical Computer Science that recently got reorganized into the Department of Information and Computer Science. I am grateful for Professors Torsten Schaub (University of Potsdam) and Wolfgang Faber (University of Calabria) for their helpful comments on the manuscript and for Professor Vladimir Lifschitz (University of Texas at Austin) for kindly agreeing to be the opponent.

I would also like to thank my colleagues and friends, both those who work in the laboratory and those who are elsewhere, as well as my family and relatives. In particular, I would like to thank Elina for having a commendable amount of patience for dealing with a logician.

My work was funded by the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Academy of Finland (Project numbers 122399, 211025, 53695, and 43963).

..

1 INTRODUCTION

Answer set programming (ASP) [130, 15, 60, 114, 143] is a way to solve problems using mathematical logic: we express our problem using some declarative and decidable logical language and then use a general purpose inference engine to find the solutions. ASP differs from traditional programming in that we do not write algorithms to manipulate data but instead define the properties of a correct answer. ASP has been used to solve problems such as AI planning [82], product configuration [177, 187], and bounded model checking [95, 96].

For example, consider a puzzle by Raymond Smullyan [176] about the Island of Knights and Knaves where every person either tells the truth (a knight) always or lies (a knave):

A stranger met two inhabitants of the island. "At least one of us is a knave," said the first one. Are they knights or knaves?

The ASP approach to solve this puzzle is to express it using some logical language. If we choose classical propositional logic to do it, we get the one-line encoding:

$$T_A \leftrightarrow (\neg T_A \vee \neg T_B) \quad (1.1)$$

where T_A denotes that A is truth-teller and T_B denotes the same for B . We can now find the models of (1.1) by either computing them by hand or we can use one of the many existing implementations of propositional logic to do it. Either way we find out that there is exactly one model: $\{T_A, \neg T_B\}$: the person who answered is a knight and the other one is a knave.

This small toy example illustrates the basic operating principle of ASP: encode the problem using logic so that its solutions correspond to the models of the encoding and then let some black box compute them. As the name "answer set" hints at, we usually represent the models as sets of atoms. The general method of using programs to describe the solutions is called *declarative programming* [122].

We used propositional logic as our language in the example and nothing prevents us from using it with larger problems. However, in practical problems we often meet properties that are cumbersome to model under classical logic so most existing ASP systems operate under some other semantics. Most of them are based on some variant of the *stable model semantics* [84] of logic programming.¹

Conceptually ASP is close to constraint programming [123] (CP) that operates under the same declarative principles. In a CP program we have a set of variables and a set of possible values for every variable, and we want to find a way to assign a value for every variable such that the requirements of the problem are satisfied. In fact, we could even fit ASP

¹The current ASP implementations include Smodels [148], DLV [111], Assat [119], clasp [79] Cmodels [5], NoMoRe [2], ASPPS [52], and SAG [120]. Of these ASPPS is based on extended propositional satisfiability and the others are founded on the stable model semantics.

into the CP framework by defining each atom to be a variable with two possible values, true and false, and expressing the semantics of a program in terms of CP constraints. There are practical reasons why we do not want to do that. The ASP and CP semantics are different enough that it is cumbersome to express one in terms of another and they have different strengths and weaknesses.

1.1 LOGIC PROGRAMMING AND ASP

The origins of logic programming [121] lay in the article *The Semantics of Predicate Logic as a Programming Language* by van Emden and Kowalski [204] that was published in the Journal of ACM in 1976. In it they gave a formal semantics for the Prolog language [33]. The semantics was based on the predicate calculus and it expressed Prolog query evaluation as predicate logic theorem proving. Prolog is Turing-complete,² which means that we can solve any solvable problem with it, and it is still the best-known logic programming language.

Answer set programming takes a different approach.³ Instead of having a language that can solve everything, we use a special purpose language that we can use to solve the specific problems that we are interested in elegantly and efficiently.

In particular, many if not most **NP**-complete problems [78, 152] have simple and natural encodings as ASP programs. This class is the most famous complexity class. Intuitively, it contains problems where finding a solution is difficult but checking whether a given solution candidate is correct is easy.⁴

1.1.1 Properties of an ASP Language

A special purpose language should give good support for writing programs in its application domain. ASP is best suited for problems for which it is difficult to write an efficient traditional algorithm but whose correct solutions are easy to define. Three important properties that an ASP language should have are:

1. possibility for uniform encodings;
2. syntax expressive enough to facilitate compact and intuitive encodings; and
3. a semantics that gives intuitively correct answers.

²A language is Turing-complete if it is possible to simulate the computations of an arbitrary Turing machine using a program written with it. See Section 6.3 for one possible formal definition of Turing machines.

³There has been some work [11, 25] on Turing-complete ASP formalisms, but the actual implementations are firmly entrenched in some lower complexity class.

⁴See Section 6.1 for further information on complexity classes and their formal definitions.

Uniform Encodings

An ASP program P is a *uniform encoding* for a problem [171, 60, 130] if we can use it unaltered for solving all instances of the problem. We get a solution for a given instance by combining P with a set of facts that describes the instance and then finding an answer set of the combined program. A uniform encoding is much more convenient in practice than a non-uniform one. Not all logical formalisms enable uniform encodings, for example, with propositional logic we usually have to create a separate encoding for every problem instance.

Expressive Syntax

When we examine **NP**-complete problems [78], we find that their definitions often contain similar elements. For example, consider the three well-known graph problems:

- **VERTEX COVER**: does a graph have a k -element subset of vertices such that least one end of each edge is in it;
- **CLIQUE**: does a graph have a complete k -vertex subgraph; and
- **INDEPENDENT SET**: does a graph have a k -element subset such that no two vertices in it are connected by an edge.

The common feature in these three problems is that we are interested in finding a subset of a specified size that satisfies the constraints of the problem. We want that our language is strong enough that we can express this property and other similar common ones concisely.

Clean Semantics

The semantics of an ASP language should be simple enough that a programmer can easily understand what the answer sets of a program will be. This characterization is inherently subjective—different people find different things intuitive.

Also, if the basic semantics is simple, it is more straightforward to write a solver for it. The internal data structures can be made simpler and more efficient. This is important since ASP allows us to have simple programs for hard problems. A solver has to be efficient enough that we can find the answer sets in reasonable time.

1.2 CARDINALITY CONSTRAINT PROGRAMS

In this work we examine ASP in the context of the stable model semantics for cardinality constraint programs (CCP). The language is expressive enough to admit concise encodings for many **NP**-complete problems [198] while still having simple semantics. We consider both theoretical properties of the formalism and the practice of writing ASP encodings for different types of problems. We will use the term “stable model” when we discuss cardinality constraint programs specifically and “answer set” when discussing ASP in general.

Cardinality constraint programs extend normal logic programs. A normal program consists of rules of the form:

$$H \leftarrow L_1, \dots, L_n$$

The intuition is that if all literals L_i in the *body* of the rule are true, then the atom H in the *head* has to be true, also. What this means precisely depends on the semantics we use.

We can express a large number of problems nicely using normal logic programs, but sometimes we run into conditions that cannot be expressed compactly. The problem of finding a subset of a specified size is one of them and a straightforward encoding for selecting k items out of n needs $\binom{n}{k}$ rules to express it.⁵

In cardinality constraint programs we extend the syntax of normal programs so that we get concise encodings for more problems. The extensions allow us to express, among other things:

- disjunctive conditions in rule bodies;
- conjunctions and disjunctions⁶ in the rule head; and
- existential and universal quantification over finite relations in rule bodies.

The two most important additions to the syntax are cardinality atoms and conditional literals. A cardinality atom has the form:

$$L \{l_1, \dots, l_n\} U$$

and it is true if the number of true literals in the set $\{l_1, \dots, l_n\}$ is between the lower (L) and upper (U) bounds. A conditional literal has the form $a(X) : d(X)$ and it denotes the set of atoms $a(X)$ for which it holds that $d(X)$ is true.⁷ For example, the cardinality atom:

$$k \{X.in-subset(X) : in-set(X)\} k$$

expresses the condition that a subset has exactly k elements when the predicate $in-set(X)$ defines what elements belong to the set and the predicate $in-subset(X)$ defines the subset. The notation $X.$ tells us that X is *local* to the conditional literal and its scope does not extend outside it.

Cardinality atoms are just one of the possible ways to extend the normal logic programs.⁸ They have an intuitively clean semantics and

⁵When we have a set of atoms $E = \{e_1, \dots, e_n\}$, the rule:

$$k\text{-true} \leftarrow e_1, \dots, e_k, \text{not } e_{k+1}, \dots, \text{not } e_n$$

states that there are k true atoms if the atoms $\{e_1, \dots, e_k\}$ are true and all others are false. However, if e_1 is false and e_n is true, this rule does not recognize it. We need a similar rule for every k element subset of E .

⁶However, our semantics is not disjunctive logic programming [60] in the technical sense of the term.

⁷The formal definition that is presented in Section 3.7 is a bit more complex.

⁸We will examine several others in Chapter 7.

they can be used to model a great number of constructs that occur in practical problems.

Most of the earlier work on programs with cardinality atoms [146, 179, 145, 175] has concentrated on variable-free programs and variables have received only little attention [193]. In this work we examine variables and interpreted function symbols with more detail.

Having both an expressive language and a semantics that can be implemented in a straightforward way are two goals that are difficult to reconcile, since adding more features to the language adds complications to the solver design.

In this work we take a two-phase approach. We first define a simple basic language, and then define the full language as its extension. The semantics of the extensions are defined via translations back to the basic language. This approach makes it possible for us to have both an expressive language and a relatively simple implementation.

1.3 SCIENTIFIC CONTRIBUTIONS

We define the syntax and semantics of cardinality constraint programs that is an answer set programming language including cardinality atoms and conditional literals.

We first define a simple basic language and the constructs of the full language are defined as transformations back to the basic language. The basic language is defined so that it is possible to do the transformations on the level of programs with variables in such a way that it does not add new atoms into the instantiation of the program.⁹

The semantics is initially defined for ground programs and a program with variables is handled by first instantiating it into a ground program. The novelty of our approach is that we allow instantiation with respect to an arbitrary universe instead of using the standard Herbrand universe of program. This makes it possible to incorporate interpreted function symbols directly into the semantics. We define a standard interpretation that extends the Herbrand instantiation of a program by adding arithmetic operations as well as interpreted constants that can be used to parametrize programs.

We identify a subclass of the language, ω -restricted programs, that is decidable even when we use the Herbrand interpretation of the function symbols.¹⁰ We prove this by showing that we need to consider only a finite subset of the infinite Herbrand instantiation when computing the answer sets of a program and we show how we can compute this relevant instantiation. We examine how we can interpret the instantiation process in terms of relational database operations analogously to the Datalog [26]

⁹More precisely, we do not add atoms that may increase the search space of program. In some transformations we introduce a couple of auxiliary atoms that have the property that after we have instantiated the program, we can decide their truth values in linear time with respect to the size of the instantiation and thus we can remove them from the program with a small amount of preprocessing.

¹⁰The full language is not decidable under the Herbrand interpretation.

language.

We analyze the computational complexity of the language. We examine the complexity of both instantiation and model existence for four subclasses of ω -restricted programs. Additionally, we examine the functional problem of finding an answer set. We introduce an abstract framework for comparing the amount of effort that it takes to compute answer sets of different programs. This framework applies the concept of backdoors [209] from the propositional satisfiability research to logic programs.

We examine how we can create a practical instantiator for ω -restricted programs under the standard interpretation. The general architecture of the instantiator is based on compiler technology [1]. There is a front-end that translates the user programs into an internal representation and a back-end that instantiates the program. We present an instantiation algorithm that is based on the relational data model and prove that it is correct.

We present several practical examples for using cardinality constraint programs to model different problems. The encodings are uniform [130] and are based on the generate and test method [141]. There is a complete example on how we can create efficient encodings by reducing the search space that a solver has to examine while searching for answer sets.

We also present a framework for integrating answer set programming into traditional programming systems where a solver is used as an oracle [200, 152] that can be called from a conventional program.

Finally, we give an extended application example by showing how we can express planning problems with cardinality constraint programs. We use an extended version of the action language \mathcal{B} [88] as the planning formalism and give a systematic translation from it to logic programs and prove that it is correct.

We discuss several issues that have to be considered when designing planning encodings that allow parallelism in domains that are not inherently parallel and briefly examine how the translation can be extended to cover constructs that are present in different action languages.

We give three concrete examples for planning encodings. Two of them are relatively simple and we can directly utilize the translation. The third example shows how we can use domain-specific knowledge to optimize the encoding so that it is more efficient in practice.

1.4 RELATED WORK

1.4.1 Aggregates

The cardinality atoms are a form of aggregate literals [107] that combine the truth values of a set of literals into one aggregate value. The purpose of aggregates is to allow the programmer to write programs that are more concise and intuitive than a corresponding normal logic program would be. There has been much research on different types of aggregate literals [107, 42, 43, 129, 69, 126, 183, 154, 182, 65, 67]. We will discuss these semantics in Section 7.1.6 after we have defined our basic semantics.

1.4.2 Conditional Literals

Conditional literals [197] allow us to define sets of literals in a concise way. Similar syntax has been used for defining aggregates [42, 126] as well as in defining parametric connectives [155]. We will examine these constructs in more detail in Section 7.1.4.

1.4.3 Function Symbols

Standard arithmetic operators are the most commonly used function symbols in ASP systems. They are often seen as notational shortcuts [111] that could be replaced by defining explicitly suitable predicates.

The parametric external predicates of Ianni et al. [19] can be used to create behavior similar to our interpreted functions. An external predicate symbol $\#p$ is associated with an oracle function f_p that tells whether the external predicate is true or not. For example, with an atom $\#sum(1, 2, 3)$ the function $f_{sum}(1, 2, 3)$ returns true and with $\#sum(1, 2, 4)$ it returns false. An oracle function is essentially the same as our interpretation function. The main difference between these approaches is that function symbols can be nested more easily than external predicates.

While logic programming formalisms with function symbols are generally undecidable, the finitary programs [11, 10, 6] of Bonatti are decidable with respect to ground queries. We will examine those programs in Section 7.1.7.

1.4.4 Computational Complexity

Dantsin et al. [37] analyzed the computational complexity of normal and disjunctive logic programs, while Faber and Leone [67] examined the complexity of aggregates. In this work we extend these results to the ω -restricted programs. In the complexity proofs we use a direct reduct from Turing machines. A similar Turing machine encoding for ASP has been presented by Marek and Remmel [127]. Our encoding is a bit simpler than the previous one.

1.4.5 Language Translations

The expressive power of an ASP system can be increased by defining a new language whose programs are then translated into original language programs. In this work we use this concept in two ways: the full language is translated into the basic language and an AI planning action language is translated into the full language.

The idea of building a more expressive language on top of a less expressive one is well-known in ASP research. For example, the approach has been used for express nested expressions [118], diagnosis computation [53], disjunctive programs with inheritance [14], multiple different preference semantics [41, 58, 13], and even logical puzzles [74]. Also, the template programs of Calimeri and Ianni [20] work in essentially the same way.

1.4.6 Instantiation

When we examine the implementation of CCPs, we concentrate on the problem of instantiating a program. Faber et al. [66, 36, 35] have done work on applying deductive database techniques for ASP instantiation, and the XSB team [161, 24] has integrated ASP solvers into a conventional logic programming system that instantiates a program as a part of model computation, and Gebser et al. [81] have implemented an instantiator for λ -restricted programs.

We will examine this body of work in Section 8.4.3.

1.4.7 Planning

Our largest programming example is a translation from an action language [88] to CCPs. Several similar translations [136, 56, 49] have been presented in literature. We will examine these translations in Section 10.

1.4.8 Semantics Integration

Aggregates are one way to add more expressive power to an ASP language. Another way to make the systems more expressive is to combine more than one formal semantics into a single framework. Apart from a brief informal excursion to oracles in Section 9.3 we do not examine this approach in this work.

Several examples of semantics integration have been published. Eiter et al. [61] give a general framework for that and use it to combine description logics with disjunctive logic programs [62]. Similarly, there has been work on combining ASP with traditional constraint programming techniques [7, 138].

1.4.9 Other Semantics

In our work we examine ASP in the form of extensions to the stable model semantics of normal logic programs. This is not the only option and some other semantics have been used for ASP. For example, East and Truszczyński [52] add propositional schemata to propositional satisfiability and Denecker et al. [44, 211] work with the logic of inductive definitions. Pearce et al. [153, 149] examine ASP in the context of equilibrium logic.

1.5 OUTLINE OF THE WORK

Now we will examine how this work is structured.

Chapter 2. Here we will define the basic syntax for cardinality constraint programs. The basic building blocks of logic programs are predicate symbols, terms, atoms, literals, and rules. The features specific to CCPs are conditional literals, cardinality atoms, and choice rules.

Chapter 3. Here we define the stable model semantics for CCPs. We first examine ground programs and then extend the definition to cover also variables. The semantics is defined in terms of a reduct where we start with a set of atoms that we assume are true and simplify the program with respect to that set. If the logical closure of the simplified program coincides with our set of assumptions, we have found a stable model.

We add variables to our programs via Herbrand instantiation. A rule with variables stands for all the variable-free instances that we can generate by substituting the variables with terms of the Herbrand universe of the program.

We also examine how we can add interpreted function symbols to our semantics. We leave the realm of Herbrand terms and introduce a way how we can use function symbols to perform some actual computation during the program instantiation.

Chapter 4. The problem with Herbrand instantiation is that it is infinite if a program has any non-constant function symbols in it and the question whether there exists any stable model becomes undecidable.

Here we define a subclass of CCPs that have the property that only a finite part of the instantiation is relevant and consequently the stable models are finite and decidable. The program class is the ω -restricted programs. Their general idea is that the program is divided into two parts: domain and non-domain. In most cases the non-domain predicates are used to encode the actual problem itself and the domain predicates give all relevant variable bindings for the problem instances.

Chapter 5. Most of this technical section is devoted on proving that the ω -restricted CCPs are decidable. In Section 4 we define ω -restriction in terms of ω -stratification. Most nonempty CCPs have an infinite number of ω -stratifications but we show that they are all equivalent in the sense that they lead to the same set of domain predicates. We define the concept of related instantiation that allows to discard irrelevant rules from the Herbrand instantiation and prove that the relevant instantiation is finite for all ω -restricted programs and also give a simple algorithm for computing them.

We conclude this section by examining how we can treat instantiation as a database operation.

Chapter 6. In this section we examine the computational complexity of ω -restricted CCPs. It turns out that they are very complex. We define two different problems, INSTANTIATION and MODEL. The first one examines how difficult it is to instantiate a program and the second one asks if a program has any stable models at all.

We show that if we do not use non-constant function symbols in our programs, then the stable model semantics for ω -restricted programs is as easy or complex as it is for normal logic programs. On

the other hand, adding function symbols causes potentially double-exponential increase in complexity.

We also examine the function version of MODEL where we want to compute the models and not only determine their existence.

Chapter 7. Here we define the full CCP language. We define the new features as transformations to the basic language. In this section we also examine the design criteria of the language and compare it with related work.

Chapter 8. In this section we examine the issues that arise when we implement an instantiator for CCPs. We describe the structure of an instantiator and present one simple algorithm that can be used to compute the relevant instantiation. The final instantiation is in a form that can be used together with the *smodels* solver.

Chapter 9. In this section we present a general programming methodology for ASP and give several examples of how it can be used. The encodings that we use are uniform, which means that we have one program that expresses the constraints of the problem domain and the specific problem instances are defined using simple sets of facts. The encodings are based on the generate-and-test method.

We also examine how we can use ASP solvers as a part of ordinary programs. The idea is that we treat them like oracles that give answers to **NP**-queries.

Chapter 10. We examine one ASP application area in detail, namely, planning. We give a translation from a planning domain specification language into CCPs. We also examine three simple planning domains to see which kind of considerations we have to make when encoding them.

2 CARDINALITY CONSTRAINT PROGRAMS

Our approach is that we define the cardinality constraint logic programs (CCPs) in two stages: a simple basic language and a full language that is built on it. The idea is that the basic language is simple enough to allow us to analyze its properties and to implement it efficiently. Then, the full language adds more expressive power that makes it easier to design programs. After the programmer has written an encoding with the full language, the ASP system will automatically translate it into the corresponding basic language program that is then given to a solver.

In this section we define the syntax of the basic language. These definitions are based on the one presented in [193] but the notations are simplified and changed to conform with [194].

We will examine the design criteria of the language in detail in Section 7.1. The most important criterion is that we want to be able to do the translation from the full to the basic language on the level of programs with variables, so we include enough features to the basic language to allow us to do this.

The three most important additions to the syntax of normal logic programs are cardinality atoms that allow us to do basic computation based on numbers of true literals, conditional literals that allow us to define sets of basic literals in a compact manner, and choice rules that we use to generate the answer set candidates.

We will introduce several different subclasses of cardinality constraint programs. Table 2.1 contains a list of those subclasses as well as a short verbal description. These classes will be defined formally when we meet them.

2.1 SYNTAX

The syntactic elements that we use are terms, atoms and basic literals, conditional and cardinality literals, and rules.

2.1.1 Terms

A *term* is either a *variable* or an *m*-ary *function term* $f(t_1, \dots, t_m)$ where f is an *m*-ary function symbol and t_1, \dots, t_m are terms. A 0-ary function symbol is also called a *constant*. Constants and variables are *atomic terms* while terms of the form $f(t_1, \dots, t_m)$ where $m > 0$ are *compound terms*.

We use the standard logic programming convention that all variables start with an upper case letter and all other terms are function symbols. Thus, X , Y , and $Cost$ are variables while a , c , and 1 are constants. A term is *ground* if it does not contain any variables.

When we have a function symbol that corresponds to some standard mathematical function, we will often write the term using standard mathematical notation. Thus, we may use $X + Y$ instead of $+(X, Y)$.

Language	Abbr.	Description
Basic programs	CCP	Programs formed using the basic constructs
Full programs	FCCP	Programs using the full cardinality literal syntax
Extended programs	ECCP	The full language that is further extended with strong negation and weight literals.
Augmented programs	ACCP	Basic programs that are extended with integral ranges
Simple programs	SCCP	Basic programs where all cardinality atom bounds are integers.
Proper programs		Programs with a finite number of predicate symbols
ω -Restricted programs		A decidable subclass of CCPs where each variable that occurs in a rule has to also occur in a domain predicate in its body.
Positive programs		Programs without any kind of negations in them.
Stratified programs		Programs that do not have negative recursion on the level of predicate symbols
Locally stratified programs		Programs that do not have negative recursion on the level of atoms

Table 2.1: Language glossary

2.1.2 Atoms and Literals

An *atom* is of the form:

$$p(t_1, \dots, t_n)$$

where p is an n -ary *predicate symbol* (denoted p/n) and the *arguments* t_1, \dots, t_n are terms. We will use A to denote an arbitrary atom when we are not interested in its arguments, and $\text{pred}(A)$ to denote the predicate symbol of A . The symbol \top denotes a special atom that is always true and \perp denotes an atom that is always false.

A *basic literal* is either an atom A or its negation $\text{not}(A)$. The former are called *positive literals* and the latter *negative literals*. We usually leave out the parenthesis from the negative literals when writing CCPs. We use L to denote a basic literal and \bar{L} its complement¹ when their arguments are not relevant.

2.1.3 Conditional Literals

A *conditional literal* \mathcal{L} is of the form:

$$X.L : A \tag{2.1}$$

where the *main literal* L is a basic literal, the *condition* A is an atom, and X is a set of *local variables*. If $X = \emptyset$, we write (2.1) simply as $L : A$

¹So $\bar{\bar{A}} = \text{not}(A)$ and $\overline{\text{not}(A)} = A$.

and if X is a singleton set, we leave out the braces surrounding it. For example, $p(Y) : \top$, $X.a(X) : b(X)$, and $\{X, Y\}.q(X, Y) : d(X, Y)$ are all conditional literals. We will often write a conditional literal of the form $\emptyset.L : \top$ as L . These uses may be detected by context: if a basic literal occurs in a position where a conditional literal should occur, then it is a shorthand for the conditional literal.

The intuition is that the condition A controls when the literal L is included in a rule: if A is true, then L is taken in, and if not, L is discarded. We can think that a non-ground literal $\{X\}.p(X) : d(X)$ represents the set of atoms $\{p(a) \mid d(a) \text{ is true}\}$. We will give a formal definition for this in Section 3.7. A ground conditional literal is essentially equivalent to a conjunction $L \wedge A$ in its standard usage. However, we will see a language extension in Section 7.3.5 that makes it behave like an implication $A \rightarrow L$.

For all conditional literals $\mathcal{L} = L : A$ we use $lit(\mathcal{L}) = L$ and $cond(\mathcal{L}) = A$ to denote the main literal and the condition, respectively.

2.1.4 Cardinality Atoms and Literals

A *cardinality atom* C is of the form:

$$\text{Card}(b, S) \tag{2.2}$$

where the *bound* b is a term and S is a set of conditional literals. The intuition is that C is true if at least b literals in S are true. The set of positive conditional literals in S is denoted by $\text{pos}(C)$ and the set of negative conditional literals by $\text{neg}(C)$. A *cardinality literal* \mathcal{C} is either a cardinality atom C or its negation $\text{not}(C)$.

A cardinality atom is *simple* if b is an integer. For example, the $\text{Card}(2, \{X.a(X) : d(X)\})$ is simple but $\text{Card}(t + 1, \{X.a(X) : d(X)\})$ is not.

We will often write cardinality literals using the more compact syntax of the SMOLELS system [148] where $\text{Card}(b, S)$ is denoted as $b \ S$. For example, $\text{Card}(1, \{a, \text{not } b\})$ is written as $1 \ \{a, \text{not } b\}$.

2.1.5 Rules

Cardinality constraint programs have two kinds of rules, basic and choice rules. A *basic rule* is of the form:

$$A \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n \tag{2.3}$$

where the *head* A is an atom and \mathcal{C}_i in its *body* are cardinality literals. The rule (2.3) encodes the fact that if all literals in the body are true, then the head must also be true. If the body is empty ($n = 0$), then we call a basic rule *fact* since its head has to be always true. If there are no negative cardinality literals in the rule body, the rule is *positive*.

A *choice rule* has the form:

$$\{A_1, \dots, A_m\} \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n \tag{2.4}$$

where A_i are atoms and C_i cardinality literals. The intuition is that if the rule body is true, then any subsets of atoms in the head may be true. When defining the stable model semantics, we require that every atom that is in a stable model has to have a justification for it. An applied choice rule justifies its head. If an atom does not occur in the head of any rule with a satisfied body, it has to be false.

A rule with an empty head is a notational shortcut for having a new atom f as the head as well as an additional cardinality literal $\text{Card}(1, \{\text{not } f\})$ in the rule body. For example, the rule

$$\leftarrow \text{body}$$

becomes

$$f \leftarrow \text{body}, \text{Card}(1, \{\text{not } f\}) .$$

The effect of such a rule is to act as a constraint on the models of a program. If the body of the original rule is satisfied, then the model candidate is rejected.

We treat the special false atom \perp as if there was a rule:

$$\leftarrow \perp$$

in the program.

2.1.6 Cardinality Constraint Programs

A *basic cardinality constraint program* (CCP) P is a (possibly countably infinite) set of rules.

We use $\text{Atoms}(P)$ to denote the set of atoms that occur in anywhere in P and $\text{Preds}(P)$ to denote the set of its predicate symbols. For each predicate symbol $p \in \text{Preds}(P)$ we use $P|p$ to denote the set of rules where p occurs in the head.

A CCP P is *positive* if all literals (basic and cardinality) in it are positive, *simple* if all cardinality literals are simple, and *proper* if $\text{Preds}(P)$ is finite.

In this work we will mostly consider only proper CCPs. This is not an essential limitation since we can always transform a non-proper program P into a proper one: if our program has an infinite number of predicate symbols $p_0/0, p_1/0, \dots$, we can replace them with a new predicate symbol $p/1$ and constants t_0, t_1, \dots . Wherever we have an atom p_i in the program, we replace it with the atom $p(t_i)$. The construction works in a similar way even if the predicate symbols p_i have higher arities.

2.2 NOTATIONAL CONVENTIONS

In this section we define auxiliary notations for rules as well as a notational shortcut for writing cardinality atoms that correspond to basic literals of normal logic programs.

Notations for Rules

We define some functions that allow us to identify basic literals that are occurring in a rule. If $R = H \leftarrow C_1, \dots, C_n, \text{not } C_{n+1}, \dots, \text{not } C_{n+m}$ is a basic rule. Then:

$$\begin{aligned} \text{head}(R) &= \{H\} \\ \text{body}_{\mathcal{L}}^+(R) &= \{\mathcal{L} \mid \mathcal{L} \in \text{pos}(C_i) \text{ for some } i \in [1, n]\} \\ \text{body}^+(R) &= \{\text{lit}(\mathcal{L}) \mid \mathcal{L} \in \text{body}_{\mathcal{L}}^+(R)\} \\ \text{body}_{\mathcal{L}}^-(R) &= \{\mathcal{L} \mid \mathcal{L} \in \text{neg}(C_i) \text{ for some } 1 \leq i \leq n+m\} \cup \\ &\quad \{\mathcal{L} \mid \mathcal{L} \in \text{pos}(C_i) \text{ for some } n+1 \leq i \leq n+m\} \\ \text{body}^-(R) &= \{\text{lit}(\mathcal{L}) \mid \mathcal{L} \in \text{body}_{\mathcal{L}}^-(R)\} \end{aligned}$$

If $R = \{H_1, \dots, H_n\} \leftarrow \text{body}$ is a choice rule, then $\text{body}_{\mathcal{L}}^+(R)$, $\text{body}_{\mathcal{L}}^-(R)$, $\text{body}^+(R)$, and $\text{body}^-(R)$ are defined as above and:

$$\text{head}(R) = \{H_1, \dots, H_n\} .$$

The body sets are defined so that all literals that occur purely positively are contained in sets $\text{body}_{\mathcal{L}}^+(R)$ and $\text{body}^+(R)$. The former contains the conditional literals and the latter their main literals. Literals that are under at least one negation symbol are put into the sets $\text{body}_{\mathcal{L}}^-(R)$ and $\text{body}^-(R)$. For example, when we have the rule R :

$$H \leftarrow \text{Card}(1, \{A : \top, \text{not } B : \top\}), \text{not Card}(1, \{C : \top, \text{not } D : \top\})$$

the sets are:

$$\begin{aligned} \text{body}_{\mathcal{L}}^+(R) &= \{A : \top\} \\ \text{body}_{\mathcal{L}}^-(R) &= \{\text{not } B : \top, C : \top, \text{not } D : \top\} \\ \text{body}^+(R) &= \{A\} \\ \text{body}^-(R) &= \{\text{not } B, C, \text{not } D\} . \end{aligned}$$

We will often write a rule $H \leftarrow C_1, \dots, C_n$ (where H is either A or $\{A_1, \dots, A_m\}$) as a pair:

$$\langle H, \{C_1, \dots, C_n\} \rangle .$$

This notation is convenient when we want to create rules using some algorithm.

The notation $\text{Atoms}(R)$ denotes the set of all atoms that occur in the rule R .

2.2.1 Notational Shortcut for Literals

It would be very cumbersome if we had to write out all cardinality literals in the rule bodies so we adopt yet another notational shortcut. A basic literal L that occurs in a rule body denotes the cardinality atom $\text{Card}(1, \{L : \top\})$. For example, the rule:

$$\text{flies}(B) \leftarrow \text{bird}(B), \text{not } \text{flightless}(B)$$

stands for the rule:

$$\text{flies}(B) \leftarrow 1 \{ \text{bird}(B) : \top \}, 1 \{ \text{not } \text{flightless}(B) : \top \} .$$

This concludes our basic language. We extend this to the full language in Section 7.

2.2.2 On Expressions, Formulas, and Variables

Before we can move on to define the semantics for the language we need to define formally the concepts of expressions and formulas. The main difference is that we can assign a formula a truth value but that is not necessarily the case for expressions.

Definition 2.2.1 *A formula is either a basic literal, a conditional literal, a cardinality literal, or a rule.*

Definition 2.2.2 *An expression is a formula or a term.*

We use the notation $\text{Var}(E)$ to denote the set of variables that occur in an expression E . An expression E is *ground* if and only if $\text{Var}(E) = \emptyset$.

If a variable is not local, we call it a *global variable*. This is a bit of a misnomer since their scope is the rule that they occur in. That is, we add an implicit foreach quantifier in front of each rule that binds all global variables in the rule. We take the approach that a variable may not occur both as a local and a global variable in a rule and we rename conflicting variables if necessary.

In the case where it is important to make a distinction between local and global variables that occur in an expression, we denote the sets with $\text{Var}_l(E)$ and $\text{Var}_g(E)$, respectively.

For each atom $A = a(t_1, \dots, t_n)$ we define the set of *first-level variables* $\text{Var}_t(A)$ as:

$$\text{Var}_t(a(t_1, \dots, t_n)) = \{t_i \mid t_i \text{ is a variable} \} .$$

For example, the first-level variables of the atom $A = a(X, Y, Y + Z)$ are $\text{Var}_t(A) = \{X, Y\}$.

3 THE STABLE MODEL SEMANTICS

In this chapter we define the stable model semantics of CCPs. We start by examining the simple positive programs and then extend it to cover all simple ground programs. Finally, we add the variables to the semantics as well as interpreted function symbols to cover the full basic language.

We define the semantics via *reducts* in a similar way how Gelfond and Lifschitz [84] defined the stable model semantics for normal logic programs. The idea is that we remove all negative literals from the program with the respect of a set of ground atoms so that in the end we are left with a positive program that has a unique *least model*. If that model happens to coincide with the atom set that we started with, then it is a *stable model* of the program.

3.1 NORMAL LOGIC PROGRAMS

A normal logic program has only basic rules and basic literals in it. We obtain the reduct P^M [84] of a program P with respect to a set of atoms M by:

1. deleting all rules where the body contains a literal not A such that $A \in M$; and
2. deleting all negative literals from the bodies of the other rules.

The set M is a stable model if and only if it is the least model of P^M .

To visualize the process we can think that M contains those things that we believe are true. If we believe that A is true, then a rule that depends on its negation is irrelevant and we can discard it. Similarly, if we believe that A is false, then a literal not A is true and can be discarded.

Next, we check if M is a justified set of beliefs. We do it by determining what atoms follow logically from our choices of beliefs. If it happens that M does not coincide with the least model of the remaining program, we either have a belief that we cannot support ($A \in M$ but not in the least model of P^M) or we do not believe in a thing that we know to be true ($A \notin M$ but in the least model).

Example 3.1.1 Consider the program P :

$$\begin{aligned} a &\leftarrow \text{not } b \\ b &\leftarrow \text{not } a . \end{aligned}$$

Let $M_1 = \{a\}$. Then the reduct P^{M_1} is:

$$a \leftarrow .$$

Its least model is $\{a\} = M_1$, so M_1 is a stable model of P . On the other hand, if we set $M_2 = \{a, b\}$, then $P^{M_2} = \emptyset$ whose least model is empty so M_2 is not stable.

3.2 SATISFACTION AND CLASSICAL MODELS

In this section we define what it means when we say that a formula is satisfied by a set of atoms. The satisfaction relation is built up starting from atoms and continuing up to rules. In the first case we consider only simple programs where all cardinality atom bounds are integers. A set of atoms that satisfies all rules of a program is its model.

Definition 3.2.1 *Let M be a set of ground atoms. Then, M satisfies a ground formula F (denoted $M \models F$) if and only if one of the following cases holds:*

1. F is an atom A and $A \in M$ or $A = \top$;
2. F is a negative literal $\text{not } A$ and $A \notin M$;
3. F is a conditional literal $\mathcal{L} = L : A$ and both $M \models L$ and $M \models A$;
4. F is a cardinality atom $\text{Card}(b, S)$ and
$$b \leq |\{\mathcal{L} \in S \mid M \models \mathcal{L}\}| ;$$
5. F is a negative cardinality literal $\text{not Card}(b, S)$ and $M \not\models \text{Card}(b, S)$;
6. F is a basic rule $H \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$ and $M \models H$ or $M \not\models \mathcal{C}_i$ for some $1 \leq i \leq n$; or
7. F is a choice rule $\{H_1, \dots, H_m\} \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$.¹

We use the standard convention where M satisfies a set S of formulas if $M \models F$ for each $F \in S$.

Definition 3.2.2 *Let P be a ground SCCP and M a set of atoms. Then, M is a model of P (denoted $M \models P$) if and only if for all rules $R \in P$, $M \models R$. The set of all models of P is denoted with $\mathbf{M}(P)$.*

A model M of P is a minimal model (denoted $\mathbf{MM}(P)$) if for all models M' of P it holds that $M' \subseteq M$ implies that $M' = M$. If P has only one minimal model, it is called the least model.

When we want to distinguish the models of Definition 3.2.2 from stable models, we call them *classical models*. This term comes from interpreting a rule $H \leftarrow L_1, \dots, L_n$ as an implication $L_1 \wedge \dots \wedge L_n \rightarrow H$.

3.3 REDUCTS

We define the reduct using a construct that is based on the one presented for extended programs in [145]. Informally, when we construct the reduct P^M we

¹A choice rule is always satisfied because we can choose whether we want to include the head of the satisfied rule into the model or not.

1. drop all rules that contain a negative cardinality literal that is not satisfied by M as well as all negative cardinality literals from the bodies of remaining rules; and
2. replace all negative conditional literals $\text{not } A : B$ where $M \models \text{not } A$ by $\top : B$ and drop all other negative conditional literals.

The intuition for handling conditional literals is that $\text{not } A : B$ is potentially true if $\text{not } A$ is true but we have to check that the condition is also true. If $M \not\models \text{not } A$, we know that $\text{not } A : B$ is not satisfied regardless of the truth value of B and we can drop it.

Definition 3.3.1 (Reduct) *Let $\mathcal{L} = L : A$ be a ground conditional literal. Then its reduct \mathcal{L}^M with respect to the set of atoms M is:*

$$\mathcal{L}^M = \begin{cases} \{L : A\}, & \text{if } L \text{ is positive} \\ \{\top : A\}, & \text{if } L \text{ is negative and } M \models L \\ \emptyset, & \text{if } L \text{ is negative and } M \not\models L \end{cases}.$$

Let $C = \text{Card}(b, S)$ be a cardinality atom. Then its reduct C^M with respect to the set of atoms M is:

$$C^M = \text{Card}(b, S^M)$$

where

$$S^M = \bigcup_{\mathcal{L} \in S} \mathcal{L}^M.$$

Let $R = \langle H, \{C_1, \dots, C_n, \text{not } C_{n+1}, \dots, \text{not } C_{n+m}\} \rangle$ be a ground basic rule. Then, its reduct R^M is the set:

$$R^M = \begin{cases} \langle H, \{C_1^M, \dots, C_n^M\} \rangle, & \text{if } M \models \text{not } C_i \text{ for all } n < i \leq n+m \\ \emptyset, & \text{otherwise} \end{cases}.$$

Let $R = \langle \{H_1, \dots, H_i\}, \{C_1, \dots, C_n, \text{not } C_{n+1}, \dots, \text{not } C_{n+m}\} \rangle$ be a choice rule. Its reduct is the set:

$$R^M = \{ \langle H, \{C_1^M, \dots, C_n^M\} \rangle \mid H \in \{H_1, \dots, H_i\} \cap M \text{ and } M \models \text{not } C_j \text{ for all } n < j \leq n+m \}$$

The reduct of a ground SCCP P with respect to M is the set:

$$P^M = \bigcup_{R \in P} R^M.$$

The reducts of conditional literals and rules are defined to be sets since there is the possibility that we remove one of them altogether.

Example 3.3.1 *Let $M = \{a\}$ and three conditional literals be:*

$$\begin{aligned} \mathcal{L}_1 &= c : b \\ \mathcal{L}_2 &= \text{not } a : b \\ \mathcal{L}_3 &= \text{not } c : b \end{aligned}.$$

Then,

$$\begin{aligned}\mathcal{L}_1^M &= \{c : b\} \\ \mathcal{L}_2^M &= \emptyset \\ \mathcal{L}_3^M &= \{\top : b\} .\end{aligned}$$

Since $a \in M$, $\text{not } a$ is certainly not true and we remove $\text{not } a : b$ altogether. On the other hand, $c \notin M$ so $\text{not } c : b$ may be satisfied so we replace c with \top .

Example 3.3.2 Consider the case where $M = \{a, b\}$ and $C = \text{Card}(2, \{c : \top, \text{not } d : a, \text{not } b : e\})$. Now, $C^M = \text{Card}(2, \{c : \top, \top : a\})$.

Example 3.3.3 Consider the one-rule program P :

$$\{a\} \leftarrow 1 \{b : \top, \text{not } c : \top\}, \text{not } 1 \{\text{not } b : \top, c : \top\}$$

Let $M_1 = \{a, b\}$. Then, the reduct P^{M_1} is:

$$a \leftarrow 1 \{b : \top, \top : \top\} .$$

In the case of the negative cardinality literal we note that

$$M_1 \not\models \text{Card}(1, \{\text{not } b : \top, c : \top\})$$

so we keep the rule while removing the negated literal from it.

On the other hand, if we consider $M_2 = \{a, c\}$, we see that $M_2 \models \text{Card}(1, \{\text{not } b : \top, c : \top\})$ so the negative literal is not satisfied and $P^{M_2} = \emptyset$.

Finally, if $M_3 = \{b\}$, then $P^{M_3} = \emptyset$ even though the negative cardinality literal is satisfied since $a \notin M_3$.

All rules that belong to a reduct of an SCCP contain only positive literals. All such programs have a unique least model. Our proof is very similar to the one presented in [204] for normal programs.

Proposition 3.3.1 If P is a positive ground SCCP, then the intersection $M' = \bigcap \{M \mid M \in \mathbf{M}(P)\}$ is a model of P .

Proof. The set $\text{Atoms}(P)$ of all atoms in P is a classical model of P so the intersection is not empty and M' is well-defined.

Suppose that M' is not a model of M . Then, by Definition 3.2.1 there exists a rule R :

$$H \leftarrow C_1, \dots, C_n$$

in P where $M' \models C_i$ for all i but $H \notin M'$. Since P is positive, all cardinality atoms C_i have the form:

$$\text{Card}(b, \{A_1 : B_1, \dots, A_n : B_n\})$$

and $|\{A_j : B_j \mid M' \models A_j : B_j\}| \geq b$ for every C_i . Since for all models M of P it holds that $M' \subseteq M$, $M \models C_i$, that is, all cardinality atoms C_i are true in all models of P .

As we assumed that $H \notin M'$, there exists at least one model M of P such that $H \notin M$. However, this is a contradiction since then $M \not\models R$ so M is not a model. Thus, $M' = \bigcap \{M \mid M \in \mathbf{M}(P)\}$ is a model of P . \square

The result that a positive SCCP has a least model follows directly as a corollary of Proposition 3.3 since the intersection of the models is unique.

Corollary 3.3.1 *A positive SCCP has a least model.*

Choice rules do not contribute atoms to the least model. When the body of a choice rule is satisfied, we have a free choice of either adding the head to a model or leaving it out. If we add it, we get a model that is not minimal.

3.4 THE PROVABILITY OPERATOR

We can compute the least model of a positive program by using a one-step provability operator T_P that is defined analogously to the one presented in [204].

Definition 3.4.1 *Let P be a positive SCCP. Then, the provability operator $T_P : \text{Atoms}(P) \rightarrow \text{Atoms}(P)$ is the function:*

$$T_P(S) = \{A \mid A \leftarrow C_1, \dots, C_n \in P \text{ and } S \models C_i \text{ for all } 1 \leq i \leq n\} . \quad (3.1)$$

Given a set S of atoms, the T_P operator adds those atoms that are certainly true to the set $T_P(S)$. We do not consider choice rules here because when the body of a choice rule is satisfied, we have the option to either include the head or leave it out so a choice rule cannot derive atoms for the least model.

Proposition 3.4.1 *Let P be a positive SCCP. If $S_1 \subseteq S_2$, then $T_P(S_1) \subseteq T_P(S_2)$, that is, the T_P operator is monotone.*

Proof. Since P contain only positive literals, it holds that $S_1 \models C_i$ implies that $S_2 \models C_i$ for all cardinality atoms C_i occurring in the program. Thus, $T_P(S_1) \subseteq T_P(S_2)$. \square

When we use T_P , we start from the empty set and then repeatedly add all those atoms who occur as heads in basic rules whose bodies are satisfied by the set. Since T_P is monotone, the well-known Kleene's Fixed Point Theorem guarantees that we eventually reach the least fixed point of the operator.

In the programs that we will use in practice we reach this fixpoint after a finite number of T_P applications. However, if the program is infinite we may have to apply T_P a transfinite number of times.

We will now show that all fixpoints of T_P are models of P .²

Proposition 3.4.2 *If P is a positive SCCP and M is a fixpoint of T_P , then M is a model of P .*

²Note that the converse does not hold, there may be models that are not fixpoints.

Proof. We prove this via contrapositive by showing that if M is not a model, then it cannot be a fixpoint of T_P . By Definition 3.2.1 M is not a model of P if P contains a rule:

$$H \leftarrow C_1, \dots, C_n$$

where $M \models C_i$ for all i but $H \notin M$. However, M is not a fixpoint of T_P since $H \in T_P(M)$ so $T_P(M) \neq M$. \square

We will use the notation $\text{len}(A, P)$ to denote the number of times that the T_P operator has to be applied before A is derived into the model. Before we can define it formally we need to introduce one auxiliary notation.

Definition 3.4.2 *Let $f : S \rightarrow S$ be an operator on some set S . Then, for all subsets $S' \subseteq S$ and all ordinals i :*

1. $f(S') \uparrow 0 = S'$;
2. if i is a successor ordinal, then $f(S') \uparrow i = f(f(S') \uparrow i - 1)$; and
3. if i is a limit ordinal, then $f(S') \uparrow i = f(S'')$ where

$$S'' = \bigcup_{j < i} f(S' \uparrow j) .$$

Definition 3.4.3 *Let P be a positive SCCP, and A be an atom in the least fixpoint of T_P . Then, the derivation length $\text{len}(A, P)$ is the least ordinal i such that $T_P(\emptyset) \uparrow i \models A$.*

By Definition 3.2.1 $M \models \top$ for each set M so $\text{len}(\top, P) = 0$ for every program P .

We will next prove that the least fixpoint of T_P coincides with the least model of P .

Theorem 3.4.1 *If P is a positive SCCP, then the least fixed point of the operator T_P is the least model of P .*

Proof. Let P be a positive SCCP, M be its unique least model, and M' be the least fixed point of T_P . Since M' is a model by Proposition 3.4.2 and M is the least model, we get $M \subseteq M'$.

For proving the other direction we know that there exists some ordinal k such that $T_P(\emptyset) \uparrow k = M'$. Now, we have three possibilities:

- (i) $k = 0$. Then, $M' = \emptyset$. As M is the intersection of all models, $M = \emptyset$.
- (ii) k is a successor ordinal. Suppose that there exists at least one atom A such that $A \in M'$ but $A \notin M$. We choose A so that $\text{len}(A, P)$ is the smallest possible among such atoms. Since $A \in M'$, there exists a rule

$$A \leftarrow \text{Card}(b_1, S_1), \dots, \text{Card}(b_n, S_n)$$

where all $\text{Card}(b_i, S_i)$ are satisfied in $M'' = T_P(\emptyset) \uparrow k - 1$. As $A \notin M$, we know that $M \not\models \text{Card}(b_i, S_i)$ for some i . Since all literals are positive, this implies that there exists a conditional literal $\mathcal{L} = A' : B' \in S_i$ such that $M'' \models \mathcal{L}$ but $M \not\models \mathcal{L}$ since otherwise M would not be a model. However, in this case $A' \in M'$ and $\text{len}(A', P) < \text{len}(A, P)$ which contradicts our assumption that A has the shortest derivation length of all atoms that belong to M' but not in M . Thus, we know that $M' \subseteq M$.

- (iii) k is a limit ordinal. This case is otherwise identical to (ii) except that for each cardinality atom $\text{Card}(b_i, S_i)$ there exists some $j < k$ such that $T_P(\emptyset) \uparrow j \models \text{Card}(b_i, S_i)$ but $M \not\models \text{Card}(b_i, S_i)$.

□

Example 3.4.1 Let P be the program:

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow 1 \{a : \top\} \\ c &\leftarrow 2 \{b : \top, e : \top\} . \end{aligned}$$

We can find the least model of P by starting with the empty set and repeatedly applying T_P until we arrive to its fixpoint:

$$\begin{aligned} T_P(\emptyset) &= \{a\} \\ T_P(\{a\}) &= \{a, b\} \\ T_P(\{a, b\}) &= \{a, b\} . \end{aligned}$$

Thus, the least model of P is $\{a, b\}$. Here

$$\begin{aligned} \text{len}(a, P) &= 1 \\ \text{len}(b, P) &= 2 . \end{aligned}$$

Example 3.4.2 Consider the program P :

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow 1 \{a : b\} \end{aligned}$$

Now,

$$\begin{aligned} T_P(\emptyset) &= \{a\} \\ T_P(\{a\}) &= \{a\} = M . \end{aligned}$$

We cannot derive b since $M \not\models a : b$ as $b \notin M$.

Example 3.4.3 We will next examine a program P where we need a transfinite number of T_P applications before we get to the fixpoint:

$$\begin{aligned} a_0 &\leftarrow \\ a_1 &\leftarrow a_0 \\ a_2 &\leftarrow a_1 \\ a_3 &\leftarrow a_2 \\ &\vdots \end{aligned}$$

Every application of T_P adds one atom to the model:

$$\begin{aligned}
T_P(\emptyset) \uparrow 0 &= \{a_0\} \\
T_P(\emptyset) \uparrow 1 &= \{a_0, a_1\} \\
T_P(\emptyset) \uparrow 2 &= \{a_0, a_1, a_2\} \\
T_P(\emptyset) \uparrow k &= \{a_0, \dots, a_k\} \\
&\vdots \\
T_P(\emptyset) \uparrow \omega &= \{a_0, a_1, \dots\} \\
T_P(\emptyset) \uparrow \omega + 1 &= \{a_0, a_1, \dots\}
\end{aligned}$$

Thus, every atom belongs in the least model and we need ω steps to reach it.

3.5 THE STABLE MODEL SEMANTICS OF GROUND PROGRAMS

We are ready to formally define the stable models of ground simple CCPs. A set of atoms is a stable model of a program if it coincides with the least model of its reduct.

Definition 3.5.1 (Stable Models) *The set of atoms M is a stable model of a ground SCCP P if and only if $M = \mathbf{MM}(P^M)$.*

Next we study several example programs and their stable models.

Example 3.5.1 *Consider the program P :*

$$\begin{aligned}
\{a\} &\leftarrow 1 \{\text{not } b : \top\} \\
\{b\} &\leftarrow 1 \{\text{not } a : \top\} \\
c &\leftarrow 1 \{b : \top\} .
\end{aligned}$$

Consider the first model candidate $M_1 = \{a\}$. The reduct P^{M_1} is:

$$\begin{aligned}
a &\leftarrow 1 \{\top : \top\} \\
c &\leftarrow 1 \{b : \top\} .
\end{aligned}$$

Since $\mathbf{MM}(P^{M_1}) = \{a\} = M_1$, it is a stable model. The reduct with respect to the set $M_2 = \{b, c\}$ is:

$$\begin{aligned}
b &\leftarrow 1 \{\top : \top\} \\
c &\leftarrow 1 \{b : \top\} .
\end{aligned}$$

and $\mathbf{MM}(P^{M_2}) = M_2$ so it too is a stable model. The final stable model $M_3 = \emptyset$. The reduct is in this case:

$$c \leftarrow 1 \{b : \top\} .$$

The program has also several additional classical models that are not stable. For example, the reduct of $M_4 = \{a, b, c\}$ is

$$\begin{aligned}
a &\leftarrow 1 \{\} \\
b &\leftarrow 1 \{\} \\
c &\leftarrow 1 \{b : \top\}
\end{aligned}$$

whose least model is the empty set \emptyset .

Example 3.5.2 Consider the one-rule program:

$$a \leftarrow 1 \{ \text{not } b : a \}$$

Since b does not occur in the head of any rule, it cannot be true in the least model of any reduct and we know that it is false in all stable models³ and we are left with two stable model candidates $M_1 = \emptyset$ and $M_2 = \{a\}$. The reduct of both sets is the same:

$$a \leftarrow 1 \{ \top : a \}$$

whose least model is \emptyset . Thus, M_1 is stable but M_2 is not.

Example 3.5.3 Consider the program P :

$$\begin{aligned} \{a\} &\leftarrow \\ &\leftarrow 1 \{a : \top\} . \end{aligned}$$

As we defined in Section 2.1.5, an empty rule head is a shorthand for the construct:

$$f \leftarrow 1 \{a : \top\}, 1 \{ \text{not } f : \top \} .$$

The complete program has three classical models: $M_1 = \emptyset$, $M_2 = \{a, f\}$, and $M_3 = \{f\}$. Next, we check which ones of these are stable. The reduct P^{M_1} is:

$$f \leftarrow 1 \{a : \top\}, 1 \{ \top : \top \}$$

and $\mathbf{MM}(P^{M_1}) = \emptyset = M_1$ so it is a stable model. The reduct P^{M_2} is:

$$\begin{aligned} a &\leftarrow \\ f &\leftarrow 1 \{a : \top\}, 1 \{ \} . \end{aligned}$$

Since its least model is $\{a\}$, it is not stable. Similarly, M_3 is not stable since $\mathbf{MM}(P^{M_3}) = \emptyset$.

Thus, program P has only one stable model, M_1 , where the body of the constraint $\leftarrow 1 \{a : \top\}$ is not true.

Example 3.5.4 Let P be the program:

$$\begin{aligned} \{a\} &\leftarrow \\ b &\leftarrow \text{not } 1 \{c : a\} \\ c &\leftarrow . \end{aligned}$$

This program has two stable models, $M_1 = \{a, c\}$ and $M_2 = \{b, c\}$. The reduct P^{M_1} is:

$$\begin{aligned} a &\leftarrow \\ c &\leftarrow \end{aligned}$$

and the reduct P^{M_2} is:

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

³We will later prove this as Theorem 3.6.1.

3.6 SOME PROPERTIES OF CARDINALITY CONSTRAINT PROGRAMS

We now examine some properties of stable models of SCCPs. The first thing to note is that even though stable models are not necessarily minimal, a stable model is *supported* in the sense that all atoms in it have to occur as the head of some rule whose body is also satisfied in the model. Moreover, each atom has to have a support that has no positive recursion. We call such support a *justification*.

After that we show that SCCPs are a natural extension of normal logic programs in the sense that we have a trivial interpretation of a normal program as an SCCP.

Atom Support and Justification

We start by expressing the basic property of supporting atoms in models.

Theorem 3.6.1 *Let P be a ground SCCP and M one of its stable models. Then, for each atom $A \in M$ there exists a rule $R \in P$ such that $A \in \text{head}(R)$ and $M \models \text{body}(R)$.*

Proof. Suppose that a supporting rule R does not exist for some atom $A \in M$. Then, A cannot be derived using the T_{PM} operator so A is not in the least model of the reduct P^M and M is not stable. \square

We can strengthen this theorem by noting that we can order the atoms in a stable model M in such a way that every atom $A \in M$ comes after every atom that occurs positively in the rule that is used to derive it. This result is essentially the same that is presented for normal logic programs in [64].

Theorem 3.6.2 *Let P be a ground SCCP and M one of its stable models. Then, there exists a strict total order $<_M \subset \text{Atoms}(P) \times \text{Atoms}(P)$ such that for all atoms $H \in M$ there exists a rule $R \in P$ where:*

1. $H \in \text{head}(R)$;
2. $M \models \text{body}(R)$; and
3. for all positive cardinality atoms $\text{Card}(b, S) \in \text{body}(R)$ it holds that

$$\begin{aligned} M' \models \text{Card}(b, S), \text{ where} \\ M' = \{A \in M \mid A <_M H\} . \end{aligned}$$

Proof. Let $<_{\text{lex}} \subset \text{Atoms}(P) \times \text{Atoms}(P)$ be the strict lexicographic ordering of the atom names of P . We can define the desired strict total order $<_M$ as follows:

$$\begin{aligned} <_M = \{ \langle A, B \rangle \mid \text{len}(A, P^M) < \text{len}(B, P^M) \} \\ \cup \{ \langle A, B \rangle \mid \text{len}(A, P^M) = \text{len}(B, P^M) \text{ and } A <_{\text{lex}} B \} . \end{aligned}$$

Since both $<$ and $<_{\text{lex}}$ are strict total orders, $<_M$ is also one. Let $M_i = \{A \in M \mid \text{len}(A, P^M) = i\}$ be the set of atoms that have the derivation length of i .

Consider an atom $H \in M$. Since M is stable, H is in the least model of the reduct P^M so there is a rule R with the reduct:

$$H \leftarrow \text{Card}(b_1, S_1^M), \dots, \text{Card}(b_n, S_n^M)$$

and $M_{\text{len}(H, P^M)-1} \models \text{Card}(b_i, S_i^M)$ for all $i \in [1, n]$. This means that

$$|\{\mathcal{L} \in S_i \mid M_{\text{len}(H, P^M)-1} \models \mathcal{L}\}| \geq b_i .$$

The conditional literals in S_i that are satisfied by $M_{\text{len}(H, P^M)-1}$ have two possible forms:

1. $\mathcal{L} = A : B$. In this case $M_{\text{len}(H, P^M)-1} \models A$ and $M_{\text{len}(H, P^M)-1} \models B$, so both $\text{len}(A, P) < \text{len}(H, P)$ and $\text{len}(B, P) < \text{len}(H, P)$. Thus, $A <_M H$ and $B <_M H$ so $\{A, B\} \subseteq M'$ and $M' \models \mathcal{L}$.
2. $\mathcal{L} = \top : B$. In this case it is the reduct of a negative conditional literal $\mathcal{L}' = \text{not } A : B$ such that $A \notin M$ and $M_{\text{len}(H, P^M)-1} \models B$. As before, this implies that $B <_M H$ so $B \in M'$. Furthermore, since $M' \subseteq M$, $A \notin M'$ so $M' \models \text{not } A$ and $M' \models \mathcal{L}'$.

Thus, $|\{\mathcal{L} \in S_i \mid M' \models \mathcal{L}\}| \geq b_i$ and $M' \models \text{Card}(b_i, S_i)$ so $<_M$ is a strict total order that fulfills the conditions of the theorem. \square

Correspondence to Normal Logic Programs

In Section 2.2.1 we introduced the notational shortcut of using a basic literal L to denote the cardinality atom $\text{Card}(1, \{L : \top\})$ in a rule body. We will now demonstrate that this notation is justified because an SCCP containing only such literals and basic rules has the same set of stable models as the corresponding normal logic program.

Proposition 3.6.1 *If P is a ground normal logic program and M is its stable model, then M is a stable model of the ground SCCP P' that is obtained by substituting all literals L in rule bodies of P by the cardinality atom $\text{Card}(1, \{L : \top\})$.*

Proof. Consider a rule from P and its counterpart in P' :

$$\begin{aligned} H &\leftarrow A_1, \dots, A_n, \text{not } B_1, \dots, \text{not } B_m \\ H &\leftarrow \text{Card}(1, \{A_1 : \top\}), \dots, \text{Card}(1, \{A_n : \top\}), \\ &\quad \text{Card}(1, \{\text{not } B_1 : \top\}), \dots, \text{Card}(1, \{\text{not } B_m : \top\}) . \end{aligned}$$

When we take the reduct of the first rule with respect to some set of atoms M where $\{B_1, \dots, B_m\} \cap M = \emptyset$ we get the rule:

$$H \leftarrow A_1, \dots, A_n .$$

The operator T_P will derive the head H exactly when all atoms A_i are true.

The corresponding reduct of the CCP rule is:

$$\begin{aligned} H &\leftarrow \text{Card}(1, \{A_1 : \top\}), \dots, \text{Card}(1, \{A_n : \top\}), \\ &\quad \text{Card}(1, \{\top : \top\}), \dots, \text{Card}(1, \{\top : \top\}) . \end{aligned}$$

All the cardinality atoms corresponding to the negative literals reduced into trivially true atoms. Again, the $T_{P'}$ operator will derive H exactly when all A_i are true.

If some atom B_i is true in M , then the reduct of the normal rule is empty as it is discarded altogether. The reduct of the CCP rule will contain a cardinality atom $\text{Card}(1, \{\})$ that is always false in its body, so the rule cannot be used to derive H .

We see that the least models of both reducts will be the same. This in turn means that their sets of stable models are also the same. \square

3.7 PROGRAMS WITH VARIABLES

A rule with variables denotes the set of ground rules that can be obtained by replacing its variables with ground terms. The process where we replace the variables is called *instantiation*. As there are two types of variables, local and global, the instantiation is defined in two parts. First the local variables in conditional literals are replaced by their instantiations, and then the same is done for the global variables.

Definition 3.7.1 *A universe U is a set of ground terms. The base of an SCCP P with respect to U is the set of all ground atoms that can be formed using predicate symbols in P and terms in U .*

We start by defining the instantiation in terms of the Herbrand universe of the program. Later when we add interpreted function symbols we switch to use different universes.

Definition 3.7.2 *The Herbrand universe $U_{\mathbf{H}}(P)$ of a cardinality constraint program P is the set of all ground terms that can be constructed using the function symbols that occur in P . The simple Herbrand universe $U_{\mathbf{H}}^s(P)$ is the set of all ground terms that can be constructed using function symbols that occur as arguments to predicate symbols of P . The Herbrand base $\text{Atoms}_{\mathbf{H}}(P)$ is the base of P with respect to $U_{\mathbf{H}}^s(P)$.*

The difference between simple and standard Herbrand universes is that we do not include terms that occur as bounds in the simple one. Even though the bounds were defined to be terms, they do not occur in the atoms and we do not want to needlessly introduce them when instantiating rules. The reason why we are willing to accept this additional complexity in the Herbrand universe definition is that they allow us more flexibility in defining the bounds when we use interpreted functions. For now, we continue with the assumption that the bounds are simple integers.

Example 3.7.1 *Let P be the program:*

$$\begin{aligned} p(a) &\leftarrow \\ p(f(X)) &\leftarrow p(X) . \end{aligned}$$

Then,

$$U_{\mathbf{H}}^s(P) = \{a, f(a), f(f(a)), f(f(f(a))), \dots\}, \text{ and} \\ \text{Atoms}_{\mathbf{H}}(P) = \{p(a), p(f(a)), p(f(f(a))), \dots\} .$$

Definition 3.7.3 (Substitution) A substitution is a function $\sigma_{V,U} : V \rightarrow U$ that maps a set of variables V to a universe U . The set of all substitutions from V to U is denoted by $\text{subs}(V, U)$.

In case that V is empty, the only substitution is the empty function $\emptyset \rightarrow U$.

Definition 3.7.4 (Application) A substitution $\sigma_{V,U}$ applied to a variable v is the term:

$$v\sigma_{V,U} = \begin{cases} \sigma_{V,U}(v), & v \in V \\ v, & \text{otherwise,} \end{cases}$$

and a substitution σ applied to a function term $t = f(t_1, \dots, t_n)$ is the term $t\sigma = f(t_1\sigma, \dots, t_n\sigma)$. A substitution σ applied to an atom $A = p(t_1, \dots, t_n)$ is the atom $A\sigma = p(t_1\sigma, \dots, t_n\sigma)$, to a conditional literal $\mathcal{L} = X.L : A$ is the conditional literal $X.L\sigma : A\sigma$, and to a cardinality atom $C = \text{Card}(b, S)$ is the cardinality atom $C\sigma = \text{Card}(b\sigma, \{\mathcal{L}\sigma \mid L \in S\})$.

Whenever we apply a substitution directly to a conditional literal, we substitute only global variables. We handle local variables by *expanding* them. We replace such literal by the set of all literals that can be formed by substituting terms in the Herbrand universe for all local variables. Though, we give the definition in a more general form so that we can later use an arbitrary universe.

Definition 3.7.5 (Expansion) Let $\mathcal{L} = X.L : A$ be a conditional literal and U be an universe. Then, the expansion $\mathcal{E}(\mathcal{L}, U)$ of \mathcal{L} is the set:

$$\mathcal{E}(\mathcal{L}, U) = \{L\sigma : A\sigma \mid \sigma \in \text{subs}(X, U)\} .$$

The expansion of a cardinality atom $C = \text{Card}(b, S)$ is

$$\mathcal{E}(C, U) = \text{Card}(b, S') \text{ where} \\ S' = \bigcup_{\mathcal{L} \in S} \mathcal{E}(\mathcal{L}, U) .$$

Example 3.7.2 Let $U = \{a, b, c\}$ and $\mathcal{L} = X.p(X, Y) : q(X)$. Then, $\mathcal{E}(\mathcal{L}, U) = \{p(a, Y) : q(a), p(b, Y) : q(b), p(c, Y) : q(c)\}$.

Definition 3.7.6 (Instantiation) Let U be a universe. Then, the instantiation $\text{inst}(R, U)$ of a basic rule $R = \langle H, \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \rangle$ is the set of rules:

$$\text{inst}(R, U) = \{ \langle H\sigma, \{\mathcal{E}(C_1, U)\sigma, \dots, \mathcal{E}(C_n, U)\sigma\} \rangle \\ \mid \sigma \in \text{subs}(\text{Var}_g(R), U) \}$$

The instantiation of a choice rule $R = \langle \{H_0, \dots, H_i\}, \{C_1, \dots, C_n\} \rangle$ is the set of rules:

$$\text{inst}(R, U) = \{ \langle \{H_0\sigma, \dots, H_i\sigma\}, \{\mathcal{E}(C_1, U)\sigma, \dots, \mathcal{E}(C_n, U)\sigma\} \rangle \mid \sigma \in \text{subs}(\text{Var}_g(R), U) \} .$$

Definition 3.7.7 (Herbrand Instantiation) The Herbrand instantiation of an SCCP P is the set of rules:

$$\text{inst}_{\mathbf{H}}(P) = \bigcup_{R \in P} \text{inst}(R, U_{\mathbf{H}}^s(P)) .$$

Example 3.7.3 Let P be the program:

$$\begin{aligned} d(a) &\leftarrow \\ d(b) &\leftarrow \\ \{p(X)\} &\leftarrow 1 \{Y.q(X, Y) : d(Y)\} . \end{aligned}$$

Now, $U_{\mathbf{H}}^s(P) = \{a, b\}$. Thus, the expansion of the only conditional literal is $\{q(X, a) : d(a), q(X, b) : d(b)\}$. The complete Herbrand instantiation of P is then:

$$\begin{aligned} d(a) &\leftarrow \\ d(b) &\leftarrow \\ \{p(a)\} &\leftarrow 1 \{q(a, a) : d(a), q(b, a) : d(a)\} \\ \{p(b)\} &\leftarrow 1 \{q(a, b) : d(b), q(b, b) : d(b)\} . \end{aligned}$$

Example 3.7.4 Let P be the program:

$$\begin{aligned} s(0) &\leftarrow \\ s(f(X)) &\leftarrow s(X) \\ \{a(X)\} &\leftarrow s(X) \\ \{b(X)\} &\leftarrow s(X) \\ c(Y) &\leftarrow 1 \{X.a(X) : b(X)\}, \text{not } c(Y) . \end{aligned}$$

Since P has a non-constant function symbol, its simple Herbrand universe is infinite $U_{\mathbf{H}}^s(P) = \{0, f(0), f(f(0)), \dots\}$. Its Herbrand instantiation is also infinite as well as the expansion of $\mathcal{L} = X.a(X) : b(X)$:

$$\mathcal{E}(\mathcal{L}, P) = \{a(0) : b(0), a(f(0)) : b(f(0)), a(f(f(0))) : b(f(f(0))), \dots\} .$$

The Herbrand instantiation includes a countably infinite number of rules of the form:

$$c(f^i(0)) \leftarrow 1 \{a(0) : b(0), a(f(0)) : b(f(0)), \dots\}, \text{not } c(f^i(0))$$

where there is an infinite number of ground conditional literals in each cardinality atom.

A set of ground atoms is then a stable model of an SCCP if it is a stable model of its Herbrand instantiation.

Definition 3.7.8 (Stable Models) *Let P be an SCCP. Then, a set of atoms M is a stable model of P if and only if $M = \mathbf{MM}(\text{inst}_{\mathbf{H}}(P)^M)$. The set of all stable models of P is denoted with $\mathbf{SM}(P)$.*

Example 3.7.5 *Let P be the CCP:*

$$\begin{aligned} d(0) &\leftarrow \\ d(1) &\leftarrow \\ \{a(X)\} &\leftarrow d(X) \ . \end{aligned}$$

Then, P has four stable models: $M_1 = \{d(0), d(1)\}$, $M_2 = \{d(0), d(1), a(0)\}$, $M_3 = \{d(0), d(1), a(1)\}$, and $M_4 = \{d(0), d(1), a(0), a(1)\}$.

Example 3.7.6 *Let P be the program:*

$$\begin{aligned} \text{even}(0) &\leftarrow \\ \text{even}(f(X)) &\leftarrow \text{odd}(X) \\ \text{odd}(f(X)) &\leftarrow \text{even}(X) \ . \end{aligned}$$

Since we have a non-constant function symbol in P , its Herbrand instantiation is infinite. As the program is positive, it has a unique least model that is:

$$M = \{\text{even}(f^{2i}(0)), \text{odd}(f^{2i+1}(0)) \mid i \in \mathbb{N}\} \ .$$

Example 3.7.7 *Consider again the program P from Example 3.7.4. We can otherwise choose the truth values for $a(f^i(0))$ and $b(f^i(0))$ freely, but we get a contradiction if both of them are true for some i . In particular, every set $A \subseteq \{a(f^i(0)) \mid i \in \mathbb{N}\}$ is a stable model. Thus, P has an uncountable number of stable models.*

We conclude this section by defining three auxiliary concepts. The first one is the extension of a predicate symbol.

Definition 3.7.9 (Extension) *Let P be a CCP. Then the extension of a predicate symbol $p \in \text{Preds}(P)$ in a stable model M of P is the set:*

$$\text{ext}(p) = \{A \mid A \in M \text{ and } \text{pred}(A) = p\} \ .$$

We often leave out the predicate symbol when we write out the extension. For example, $\text{ext}(p) = \{p(a, b), p(a, c)\}$ can be written as $\text{ext}(p) = \{\langle a, b \rangle, \langle a, c \rangle\}$.

The second concept is equivalence. For the purpose of most of this work the simplest possible definition of equivalence is enough and we say that two programs are equivalent if they have exactly the same stable models. Other forms of equivalence have been proposed in the literature [116, 59, 151].

Definition 3.7.10 (Equivalence) *Two CCPs P_1 and P_2 are equivalent if and only if $\mathbf{SM}(P_1) = \mathbf{SM}(P_2)$.*

Finally, we define the notion of logical consequences.

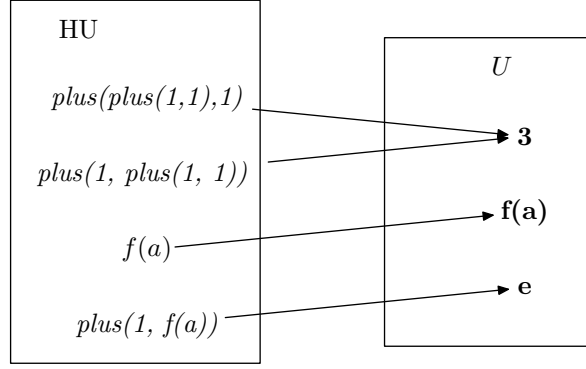


Figure 3.1: From the Herbrand universe to an arbitrary universe

Definition 3.7.11 (Logical Consequence) *Let P be a CCP. Then, a ground atom A is a*

1. *brave consequence of P if there exists a stable model M of P such that $A \in M$; and*
2. *cautious consequence of P if and only if $A \in M$ for every stable model M of P .*

3.8 INTERPRETED FUNCTION SYMBOLS

In many practical applications we want to use function symbols that are *interpreted*. For example, if we have a term $t = \text{plus}(X, Y)$ and the values of X and Y are bound to 1 and 2, respectively, we want to replace t by 3, not by the Herbrand term $\text{plus}(1, 2)$.

To allow us to have interpreted function symbols we have to leave the realm of the Herbrand terms and instantiate the program with respect to some other universe U . To see why this is necessary, examine the program P :

$$\begin{aligned} a(1) &\leftarrow \\ b(\text{plus}(X, Y)) &\leftarrow a(X), a(Y) . \end{aligned}$$

Its Herbrand universe is:

$$U_{\mathbf{H}}(P) = \{1, \text{plus}(1, 1), \text{plus}(\text{plus}(1, 1), 1), \text{plus}(1, \text{plus}(1, 1)), \dots\}$$

The constant 2 does not occur in the program at all, so if we want to use the computed value of $\text{plus}(1, 1)$, we have to add the new ground term to the program.

A standard stable model is a subset of the Herbrand base of the program and an interpreted stable model is a subset of base of the program with respect to the universe U .

We do not want to impose any restrictions on the precise nature of U : it may be the Herbrand universe, some subset of it, or an arbitrary set of elements.

3.8.1 Interpretations

An interpretation is a function that maps the terms of Herbrand universe into elements of U . When we instantiate a program, we replace all ground Herbrand terms by their interpretations.

Definition 3.8.1 (Interpretation) *Let U_1 and U_2 be universes. Then, an interpretation I is a function:*

$$I : U_1 \rightarrow U_2 \ .$$

We will extend the notation I to cover also other expressions. The idea is that we interpret all terms that occur in the expression. For example, $I(a(t_1, \dots, t_n)) = a(I(t_1), \dots, I(t_n))$ and $I(\text{Card}(b, \{L_1 \dots, L_n\})) = \text{Card}(I(b), \{I(L_1), \dots, I(L_n)\})$.

In the following examples we will use the syntax where the symbols that belong to the universe U are in bold face so that they can be differentiated from the Herbrand terms.

Example 3.8.1 *Consider the program P :*

$$\begin{aligned} s(a) &\leftarrow \\ q(0) &\leftarrow \\ s(f(X)) &\leftarrow s(X) \\ q(\text{plus}(X, 1)) &\leftarrow q(X) \ . \end{aligned}$$

The Herbrand universe

$$U_{\mathbf{H}}(P) = \{a, 0, 1, f(a), f(0), f(1), \text{plus}(0, a), \text{plus}(0, 0), \text{plus}(0, 1), \dots\} \ .$$

Suppose that we want the symbol $f/1$ to have the Herbrand interpretation that is restricted to the constant a while $\text{plus}/2$ denotes the common arithmetic addition over natural numbers. Then, we can define the universe to be:

$$\begin{aligned} U &= U_{\mathbb{N}} \cup U_f \cup \{\mathbf{e}\}, \text{ where} \\ U_{\mathbb{N}} &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}, \dots\} \\ U_f &= \{\mathbf{a}, \mathbf{f(a)}, \mathbf{f(f(a))}, \dots\} \ . \end{aligned}$$

The interpretation I is then:

$$I(t) = \begin{cases} \mathbf{0}, & \text{if } t = 0 \\ \mathbf{1}, & \text{if } t = 1 \\ I(x) + I(y), & \text{if } t = \text{plus}(x, y) \text{ and } I(x), I(y) \in U_{\mathbb{N}} \\ \mathbf{t(x)}, & \text{if } t = f(x) \text{ and } I(x) = \mathbf{x} \in U_f \\ \mathbf{e}, & \text{otherwise} \end{cases}$$

A Herbrand term that is formed using only numbers and the plus-function evaluates to the corresponding sum, terms that consist only of symbols f and a evaluate to themselves, and all other terms map to the error

term \mathbf{e} . The intuition for \mathbf{e} is that it is our way of telling which operations are undefined.

Consider the term $t_1 = \text{plus}(1, \text{plus}(1, 1))$. Its evaluation proceeds as follows:

$$\begin{aligned} I(\text{plus}(1, \text{plus}(1, 1))) &= I(1) + I(\text{plus}(1, 1)) \\ &= I(1) + (I(1) + I(1)) \\ &= \mathbf{1} + (\mathbf{1} + \mathbf{1}) \\ &= \mathbf{1} + \mathbf{2} = \mathbf{3} . \end{aligned}$$

On the other hand, the term $t_2 = \text{plus}(1, f(a))$ evaluates to the error term \mathbf{e} since $I(f(a)) = \mathbf{f(a)} \notin U_{\mathbb{N}}$.

3.8.2 Interpreted Cardinality Atoms

We first defined our semantics for simple programs where all cardinality atom bounds were integers and we introduced the concept of a simple Herbrand universe to hide the complications that arises from more general bounds. Our approach for handling cardinality atoms and literals is that:

1. we instantiate the variables that occur in a cardinality atom; and
2. interpret the resulting ground terms, including the bound.

Example 3.8.2 Consider the cardinality atom

$$\text{Card}(\text{minus}(2, 1), \{a(\text{plus}(1, 1)), b(\text{plus}(2, 1))\})$$

and an interpretation I where the functions have the usual arithmetic definitions. Then,

$$\begin{aligned} I(\text{Card}(\text{minus}(2, 1), \{a(\text{plus}(1, 1)), b(\text{plus}(2, 1))\})) \\ = \text{Card}(\mathbf{1}, \{a(\mathbf{2}), b(\mathbf{3})\}) . \end{aligned}$$

Now that we can have arbitrary terms in the bounds, we have to decide what to do with bounds that do not evaluate to integers. The simplest way to do it is to state that a cardinality atom with a non-integer bound is always false. This approach has the advantage that it makes the semantics easy to define since we can use the complete instantiation without worrying about instances where variables get substituted with non-numbers.

In Chapter 4 we will work with programs where we have to examine only a part of the instantiation and there we could take another approach where we demand that an interpretation has to evaluate all bounds to integers. This approach is more useful for practical implementations since it allows us to treat non-integer bounds as errors in the program.

We define I -satisfaction of formulas analogously to Definition 3.2.1. The only difference is now that we interpret the terms that occur in them.

Definition 3.8.2 (*I*-Satisfaction) Let M be a set of ground atoms. Then, M *I*-satisfies a ground formula F (denoted $M \models_I F$) if and only if one of the following cases holds:

1. F is an atom A and $I(A) \in M$ or $A = \top$;
2. F is a negative literal $\text{not } A$ and $I(A) \notin M$;
3. F is a conditional literal $\mathcal{L} = L : A$ and both $M \models_I L$ and $M \models_I A$;
4. F is a cardinality atom $\text{Card}(b, S)$ and

$$I(b) \leq |\{\mathcal{L} \in S \mid M \models_I \mathcal{L}\}| ;$$

5. F is a negative cardinality literal $\text{not Card}(b, S)$ and $M \not\models_I \text{Card}(b, S)$;
6. F is a basic rule $H \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$ and $M \models_I H$ or $M \not\models_I \mathcal{C}_i$ for some $1 \leq i \leq n$; or
7. F is a choice rule $\{H_1, \dots, H_m\} \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$.

We can then find the least model of a positive CCP with the one-step provability operator of Definition 3.4.1 by using the \models_I satisfaction relation.

3.8.3 Interpreted Instantiation

When we instantiate a program with respect to an interpretation I , we replace every ground term t that occurs in it with its interpretation $I(t)$.

Definition 3.8.3 (*I*-Instantiation) Let P be a CCP, U a universe, and $I : U_{\mathbf{H}}(P) \rightarrow U$ be an interpretation. Then, the *I*-instantiation of a rule $R = H \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n \in P$ is the set of rules:

$$\text{inst}(R, U, I) = \{I(H\sigma) \leftarrow I(\mathcal{E}(\mathcal{C}_1, U)\sigma), \dots, I(\mathcal{E}(\mathcal{C}_n, U)\sigma) \mid \sigma \in \text{subs}(\text{Var}_g(R), U_{\mathbf{H}}(P))\}$$

where H is either A or $\{A_1, \dots, A_n\}$ for some atom A . The *I*-instantiation of P is the set:

$$\text{inst}(P, U, I) = \bigcup_{R \in P} \text{inst}(R, U, I) .$$

Example 3.8.3 Consider Example 3.8.1. The instantiation of the third rule $s(f(X)) \leftarrow s(X)$ can be divided into two parts: those rules that do not contain the error term \mathbf{e} and the rules that have it. The first part contains the countably infinite set of rules:

$$\begin{aligned} s(\mathbf{f}(\mathbf{a})) &\leftarrow s(\mathbf{a}) \\ s(\mathbf{f}(\mathbf{f}(\mathbf{a}))) &\leftarrow s(\mathbf{f}(\mathbf{a})) \\ &\vdots \end{aligned}$$

In the second part we have rules that are of the form:

$$\begin{aligned} s(\mathbf{e}) &\leftarrow s(\mathbf{e}) \\ s(\mathbf{e}) &\leftarrow s(\mathbf{0}) \\ s(\mathbf{e}) &\leftarrow s(\mathbf{1}) \\ &\vdots \end{aligned}$$

These ground instances result from substitutions where $I(f(x)) = \mathbf{e}$.

As we see from the previous example, mixing evaluated function symbols with Herbrand interpretations generates potentially an infinite number of spurious rules. We will visit this problem and show how to resolve it in Chapter 4.

Example 3.8.4 Consider the program:

$$\begin{aligned} a(1) &\leftarrow \\ b(1) &\leftarrow \\ b(2) &\leftarrow \\ \{c(X)\} &\leftarrow b(X) \\ d(X) &\leftarrow X \{Y.c(Y) : b(Y)\}, a(X) . \end{aligned}$$

The ground instances of the final rule are:

$$\begin{aligned} d(1) &\leftarrow 1 \{c(1), c(2)\}, a(1) \\ d(2) &\leftarrow 2 \{c(1), c(2)\}, a(2) . \end{aligned}$$

3.8.4 Interpreted Stable Models

We define I -stable models analogously to standard stable models, except that we use the I -interpretation instead of the Herbrand interpretation.

Definition 3.8.4 (I -Stable Models) Let P be a CCP, U a universe, and $I : U_{\mathbf{H}}(P) \rightarrow U$ be an interpretation. Then, a set of atoms M is an I -stable model of P if and only if $M = \mathbf{MM}(\text{inst}(P, U, I)^M)$.

As we do not impose any restrictions on the nature of I , the set of I -stable models may be completely different from the set of stable models under the Herbrand interpretation. It is possible that a program that has no I -stable models has a Herbrand stable model or vice versa. The next two examples give concrete examples in both directions.

Example 3.8.5 Let I be an interpretation that maps plus to natural number addition and P be the program:

$$\begin{aligned} a(\text{plus}(1, 2)) &\leftarrow \\ b(\text{plus}(2, 1)) &\leftarrow \\ &\leftarrow a(X), b(X) \end{aligned}$$

The program P has a stable model under the Herbrand interpretation but it does not have an I -stable model since $I(\text{plus}(1, 2)) = I(\text{plus}(2, 1)) = \mathbf{3}$.

Example 3.8.6 *We now modify the program P so that it will have a stable model under I but not under the Herbrand interpretation.*

$$\begin{aligned} a(\text{plus}(1, 2)) &\leftarrow \\ b(\text{plus}(2, 1)) &\leftarrow \\ f &\leftarrow a(X), b(X) \\ &\leftarrow \text{not } f \ . \end{aligned}$$

In spite of this difference there is a connection between the Herbrand interpretation and an arbitrary one. Every ground term that occurs in the I -instantiation is the interpretation of at least one Herbrand term. This means that we can extend the definition of satisfaction to cover any atoms from the Herbrand base and say that a Herbrand atom $p(t_1, \dots, t_n)$ is satisfied in an I -stable model M if and only if $p(I(t_1), \dots, I(t_n)) \in M$.

3.8.5 On the Nature on Interpretations

We allow an interpretation to be an arbitrary function between the universes. In practice, we want to use functions that

1. can be computed efficiently; and
2. induce a unique instantiation for the program.

The reason for the first property is clear. If we want to compute I -stable models, we need to be able to compute the interpretation in a reasonable time.

The reason for the second property is not as obvious since letting the instantiation depend on the candidate stable model would allow us more expressive power. There are two main problems with this, a practical one and a theoretical one.

The practical problem is that we want to compute the stable models bottom-up by first instantiating the program and then computing the models. If a program can have more than one instantiation, we may have to instantiate it separately for each model candidate. This adds a significant amount of computational effort.

The theoretical problem is that by defining the interpretation suitably we can use it to induce extra choices to the program. When an instantiation can depend on the model candidate, we cannot guarantee that even the simplest CCPs have a unique stable model. We illustrate this with the next example.

Example 3.8.7 *Consider the program P :*

$$\begin{aligned} p(X) &\leftarrow q(X), r(X) \\ q(a) &\leftarrow \\ r(b) &\leftarrow \ . \end{aligned}$$

The Herbrand instantiation of the program is:

$$\begin{aligned} p(a) &\leftarrow q(a), r(a) \\ p(b) &\leftarrow q(b), r(b) \\ q(a) &\leftarrow \\ r(b) &\leftarrow . \end{aligned}$$

Let $U = \{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$. Let $I(M)$ be an interpretation that depends on a stable model candidate M that is defined as:

$$\begin{aligned} I(M)(a) &= \begin{cases} \mathbf{1}, & p(\mathbf{1}) \in M \\ \mathbf{2}, & p(\mathbf{1}) \notin M \end{cases} \\ I(M)(b) &= \begin{cases} \mathbf{1}, & p(\mathbf{1}) \in M \\ \mathbf{3}, & p(\mathbf{1}) \notin M \end{cases} . \end{aligned}$$

Consider the set $M_1 = \{p(\mathbf{1}), q(\mathbf{1}), r(\mathbf{1})\}$. As $p(\mathbf{1})$ is true in it, $I(M)(a) = I(M)(b) = \mathbf{1}$ and the interpretation of the Herbrand instantiation is:

$$\begin{aligned} p(\mathbf{1}) &\leftarrow q(\mathbf{1}), r(\mathbf{1}) \\ q(\mathbf{1}) &\leftarrow \\ r(\mathbf{1}) &\leftarrow \end{aligned}$$

It has the least model $\{q(\mathbf{1}), r(\mathbf{1}), p(\mathbf{1})\} = M_1$ so M_1 is I -stable.

Next, consider the set $M_2 = \{q(\mathbf{2}), r(\mathbf{3})\}$. As $p(\mathbf{1}) \notin M_2$, the interpretation is now:

$$\begin{aligned} I(M)(a) &= \mathbf{2} \\ I(M)(b) &= \mathbf{3} \end{aligned}$$

and the instantiation is interpreted as:

$$\begin{aligned} p(\mathbf{2}) &\leftarrow q(\mathbf{2}), r(\mathbf{2}) \\ p(\mathbf{3}) &\leftarrow q(\mathbf{3}), r(\mathbf{3}) \\ q(\mathbf{2}) &\leftarrow \\ r(\mathbf{3}) &\leftarrow . \end{aligned}$$

Its least model agrees with M_2 so it is also stable.

Throughout this work we will be using only interpretations where each program has only one instantiation.

3.9 THE STANDARD INTERPRETATION

We will now present a standard interpretation that we will be using in the examples of this work. Since the Herbrand universe may be different for every program, we cannot give a single interpretation that works all times but instead we give a schema for constructing it.

The basic idea is that the set of function symbols $\mathcal{F}(P)$ that occur in a program P is divided into two parts:

Function	Infix	Description
<i>plus/2</i>	$X + Y$	Addition
<i>minus/2</i>	$X - Y$	Subtraction
<i>times/2</i>	$X \times Y$	Multiplication
<i>div/2</i>	X/Y	Integer Division
<i>mod/2</i>	$X \bmod Y$	Modulus
<i>pow/2</i>	X^Y	Exponentiation
<i>abs/1</i>	—	Absolute value

Table 3.1: The arithmetic operations

1. $\mathcal{F}_{\mathbf{H}}(P)$ containing symbols with the Herbrand interpretation; and
2. $\mathcal{F}_I(P)$ containing symbols with some other interpretation.

Example 3.9.1 *Consider the program P from Example 3.8.1. There we have:*

$$\begin{aligned}\mathcal{F}_{\mathbf{H}}(P) &= \{a/0, f/1\} \\ \mathcal{F}_I(P) &= \{0/0, 1/0, plus/2\} \ .\end{aligned}$$

*The constant a and the unary function symbol f have the Herbrand interpretation while the two numeric constants and *plus* have the standard arithmetic interpretation.*

The interpretation is formed along the principle of Example 3.8.1. We will introduce a number of arithmetic operators that are evaluated in the usual way. These operators are shown in Table 3.1.

In addition to the interpreted functions we add also several related built-in predicates. These predicates implement term equality testing and other relational comparisons.

3.9.1 The Universe

The target universe $U(P)$ is composed of three parts:

$$U(P) = U_f(P) \cup U_{\mathbb{Z}} \cup U_c,$$

where the different term types are:

1. a set $U_f(P)$ containing all ground terms that can be constructed using the function symbols of $\mathcal{F}_{\mathbf{H}}(P)$.
2. a set $U_{\mathbb{Z}}$ of integers; and
3. a set U_c of additional constants.

The set $U_f(P)$ is a subset of the Herbrand universe and will usually contain at least most of the non-numeric constant symbols that occur in the program. The set U_c contains the error term \mathbf{e} . Any of the three components of the universe may be left out if desired.

Example 3.9.2 *Continuing Example 3.9.1 we have:*

$$\begin{aligned} U_f(P) &= \{\mathbf{a}, \mathbf{f}(\mathbf{a}), \mathbf{f}(\mathbf{f}(\mathbf{a})), \dots\} \\ U_{\mathbb{Z}} &= \mathbb{Z} \\ U_c &= \{\mathbf{e}\} . \end{aligned}$$

Note that there are situations where we do not want to use such a strict separation between arithmetic and Herbrand functions. For example, we may have a situation where we want to mix them. For example, if we have terms of the form $f(a, n)$ where a is a non-numeric constant and n is a number. Then we want to allow things like writing $f(A, X + Y)$ in the rules. In these cases we need to define the interpretation in some other way.

3.9.2 Interpretation Functions

We compose the interpretation I_s in parts. We define an auxiliary function I_f for every function symbol in $f \in \mathcal{F}_I(P)$ that tells how that symbol is interpreted.

Definition 3.9.1 *The standard interpretation $I_s : U_{\mathbf{H}}(P) \rightarrow U(P)$ of a CCP P is the function:*

$$I_s(t) = \begin{cases} \mathbf{f}(\mathbf{x}_1, \dots, \mathbf{x}_n), & \text{if } t = f(t_1, \dots, t_n), f \in \mathcal{F}_{\mathbf{H}}(P), \text{ and} \\ & I_s(t_i) = \mathbf{x}_i \in U_f(P) \text{ for all } i \in [1, n] \\ I_f(I_s(t_1), \dots, I_s(t_m)), & \text{if } t = f(t_1, \dots, t_m) \text{ and } f \in \mathcal{F}_I(P) \\ \mathbf{e}, & \text{otherwise} \end{cases}$$

where I_f is the interpretation function of f .

Here the idea is that the terms with the Herbrand interpretation will evaluate to themselves and for the rest of the functions the definition of I_f is responsible for handling all possible arguments. Mixed terms where an arbitrary function symbol occurs inside a Herbrand term are interpreted as errors.

We will present here an interpretation function only for addition and multiplication, the rest are defined analogously.

Numbers

We treat numbers that occur in a program as 0-ary interpreted function symbols. For each number n we define the function:

$$I_n() = \mathbf{n} .$$

For example, $I_5() = \mathbf{5}$.

Addition

The interpretation function $I_+ : U \times U \rightarrow U$ is defined as follows:

$$I_+(t_1, t_2) = \begin{cases} t_1 + t_2, & \text{if } t_1, t_2 \in U_{\mathbb{Z}} \\ \mathbf{e}, & \text{otherwise} . \end{cases}$$

$<$	$>$
\leq	\geq
$=$	\neq

Table 3.2: The standard predicate symbols

Multiplication

The interpretation function $I_{\times} : U \times U \rightarrow U$ is defined as follows:

$$I_{\times}(t_1, t_2) = \begin{cases} t_1 \times t_2, & \text{if } t_1, t_2 \in U_{\mathbb{Z}} \\ \mathbf{e}, & \text{otherwise} \end{cases}.$$

Numerically Interpreted Constants

In many cases, especially when we are working with problems that are somehow parametrized with numbers, it is useful to set the parameters in the interpretation. For example, we may have a rule:

$$\leftarrow \text{cost}(X), \text{greater}(X, k)$$

where we want to set the value of k from outside the program. A convenient way to do it is to include k in the set of interpreted functions and define I_k to evaluate to the desired value.

3.9.3 Standard Predicate Symbols

Strictly speaking standard predicate symbols are not a part of the standard interpretation since it handles only terms, but it is useful to have several relational operators available for use. The standard predicates are used like other symbols except that their extensions are fixed and depend only on the universe. Formally, we can think that they are defined by a large set of facts and they may not occur in heads of rules in the program. The standard predicate symbols are presented in Table 3.2.

The comparison predicates are defined so that they can have both numbers and other terms as arguments. If both arguments are numbers, then they are compared numerically. Otherwise, the comparison is made lexicographically based on the syntactic representation of the ground term. For example, $aa < ab$.

4 OMEGA-RESTRICTED PROGRAMS

We define the semantics of a program with global variables to coincide with the stable models of its instantiation. This has the practical advantage that it allows us to use existing solvers that work with ground programs. We compute stable models by instantiating the program and then computing the stable models of the resulting ground program. However, here we run into the problem that the Herbrand instantiation of a program is infinite if there is at least one non-constant function symbol. In these cases we cannot create the complete instantiation to find the models.

We could hope that we had some method of computing the stable models without having to compute the full instantiation. Unfortunately, this hope turns out to be false. Marek et al. [134] showed that the existence of a stable model for a normal logic programs with function symbols is undecidable and such programs are essentially a special case of CCPs.

Theorem 4.0.1 *The problem of existence of a stable model of a cardinality constraint program is undecidable in the general case.*

Proof. Marek et al. [134] showed that the stable model semantics of normal logic programs with function symbols is undecidable. By Proposition 3.6.1 each normal logic program corresponds to a CCP. \square

We are interested in finding a class of cardinality constraint programs for which we can guarantee that we can always compute the stable models and that the models themselves are finite. Moreover, we want the class to be simple enough that we can easily check whether a program belongs in it or not. We want that the check itself can be done by just examining the syntax of the rules.

The idea that we use is based on the well-known concept of *range-restriction* [142]. The basic idea of range-restriction is that every variable that occurs in a rule has to occur also in at least one positive atom in the rule body. For example, the rule

$$a(X, Y) \leftarrow b(X), c(Y), \text{not } d(X, Y)$$

is range-restricted but

$$a(X, Y) \leftarrow b(X), \text{not } d(X, Y)$$

is not since Y does not occur in a positive literal in the rule body. Range-restriction is used to bind values of variables so that we do not have to instantiate the rule for all ground terms that occur in a program.

The standard range-restriction is not strong enough for our purposes since it does not ensure finiteness of the models. For example, the program

$$\begin{aligned} s(a) &\leftarrow \\ s(f(X)) &\leftarrow s(X) \end{aligned}$$

is range-restricted but has an infinite stable model. Thus, we modify the concept a bit and introduce ω -restriction [191]. We define a set of predicate symbols, called domain predicates, in a manner that guarantees that their extensions are always finite. Every variable that occurs in a rule has to occur also in a positive domain literal, and since the domain literals have finite extensions, we know that every rule has only a finite number of ground instances that may justify atoms in stable models.

In this chapter we consider only proper programs, that is, programs that have a finite number of predicate symbols. Also, we use the Herbrand interpretation for the programs. The definitions generalize naturally to the interpreted functions.

4.1 BASIC CONCEPTS

We divide the predicate symbols of a program into two classes: *domain* and *non-domain* predicates. A predicate symbol is a domain predicate if it meets several syntactic criteria that ensure that its extension is the same in every stable model of the program. Otherwise, it is a non-domain predicate.

When a program is instantiated, the domain predicates are used to prune out rules that have provably unsatisfiable bodies. This way our instantiation stays finite and we can compute the stable models.

The domain predicates are defined by the largest grounded stratifiable [27] subset of the rules in the program. The base case is that every predicate that is defined using only ground facts is a domain predicate. Then we form a hierarchy where more complex domain predicates are defined in the terms of simpler domain predicates without using negative recursion.

We will now go through one example on an informal level before giving the formal definitions.

Example 4.1.1 *Consider the program P :*

$$\begin{aligned} a(1) &\leftarrow \\ a(2) &\leftarrow \\ b(X) &\leftarrow a(X) \\ \{c(X)\} &\leftarrow b(X) . \end{aligned}$$

Here $a/1$ is defined using only facts so we immediately know that the extension of $a/1$ is $\{a(1), a(2)\}$ in every stable model of P . Next, since $b/1$ depends only on $a/1$ and $b(X)$ is true if and only if $a(x)$ is true, we note that it too has a fixed extension and we can use it as a domain predicate.

On the other hand, the extension of the predicate symbol $c/1$ is not fixed and there are four different possibilities: \emptyset , $\{c(1)\}$, $\{c(2)\}$, and $\{c(1), c(2)\}$. Thus, we cannot use $c/1$ as a domain predicate.

In the later part of this section we will use the well-known Hamiltonian cycle problem as our running example. A Hamiltonian cycle is a path that visits all nodes of a graph without going through any node twice.

Example 4.1.2 Let P_{HC} be the following program for computing Hamiltonian cycles for undirected graphs.¹

$$\begin{aligned}
\{hc(X, Y)\} &\leftarrow edge(X, Y) \\
&\leftarrow 2 \{Y.hc(X, Y) : edge(X, Y)\}, vtx(X) \\
r(Y) &\leftarrow 1 \{r(X), initial(X)\}, hc(X, Y), edge(X, Y) \\
&\leftarrow vtx(X), \text{not } r(X) \\
edge(Y, X) &\leftarrow edge(X, Y), vtx(X), vtx(Y) .
\end{aligned}$$

This program has a stable model if and only if a graph that is defined with facts for $vtx/1$ and $edge/2$ has a Hamiltonian cycle, and the edges that belong to the cycle correspond to those atoms $hc/2$ that are true.

The first rule allows us to select any edge from the graph into the cycle. The second rule rejects those cycle candidates where two or more edges leave out of one node. The third rule computes the set of vertices that we can reach from a given initial vertex along the edges in the cycle and the fourth rule rejects those candidates that do not visit all vertices. Finally, the fifth rule ensures that all edges can be traversed in both directions.

With this encoding we do not have a specific rule for asserting that there should be an incoming edge for every vertex since the reachability predicate $r/1$ ensures that we will eventually visit every vertex.

4.2 DEPENDENCY GRAPHS

In this section we define what it means for a predicate to depend on another. Our definition for the dependency graph differs significantly from the standard construction so the theoretical results from the normal logic programs do not carry over. Also, our dependency graph is defined on the level of predicate symbols while it is often defined on the level of atoms and ground programs.

The main difference comes in handling negative dependencies. In the standard definition an atom that occurs in the head of a rule depends positively on each positive literal that occurs in the body and negatively on each negative one. If a normal program is positive, it has only one stable model where the atoms are derived straightforwardly with the T_P operator and an atom that depends positively on itself cannot justify itself into the model. If an atom depends negatively on itself, it can cause a choice point where we can either take the atom into a stable model or leave it out.

In our approach we extend the definition of a negative dependency to cover also choice rules and conditional literals. A choice rule gives us an option to either include the head in a model or leave it out. This is a similar behavior to a normal negative dependency loop, so we make the head to depend on negatively on itself.

We also add a negative dependency between the main literal and the condition of a conditional literal. The reason for this is technical. We

¹This encoding assumes that for each edge $\{u, v\}$ in the graph there exists a fact $edge(u, v) \leftarrow$ in the program.

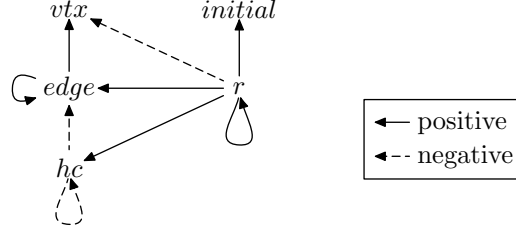


Figure 4.1: The dependency graph of the Hamiltonian cycle program.

want to be able to examine each conditional literal $\mathcal{C} = X.L : A$ in isolation. If a program contains function symbols with the Herbrand interpretation, the expansion of \mathcal{C} is infinite. However, if the extension of A is finite in each stable model, there is only a finite number of satisfiable instances $L' : A'$ in it and we can drop out all the unsatisfiable ones without affecting the set of stable models. We will define the class of ω -restricted programs in such a way that A cannot occur in the head of any rule depends on \mathcal{C} , either directly or via a longer sequence of rules. Then we can know that \mathcal{C} cannot be used to justify any new instances of A to a stable model, so we do not have to consider that possibility when examining it.² Adding a negative dependency between the main literal and the condition is a technical trick that will guarantee this result.

Definition 4.2.1 *Let P be a cardinality constraint program. Then, the one-step dependency relation $\mathfrak{D}_1(P) \subseteq \text{Preds}(P) \times \text{Preds}(P)$ is defined as follows:*

$$\begin{aligned} \mathfrak{D}_1^+(P) &= \{ \langle \text{pred}(H), \text{pred}(L) \rangle \mid R \in P, H \in \text{head}(R) \text{ and } L \in \text{body}^+(R) \} \\ \mathfrak{D}_1^-(P) &= \{ \langle \text{pred}(H), \text{pred}(L) \rangle \mid R \in P, H \in \text{head}(R) \text{ and } L \in \text{body}^-(R) \} \\ &\quad \cup \{ \langle \text{pred}(L), \text{pred}(A) \rangle \mid \text{there exists } \mathcal{L} \in \mathcal{L}(P) \text{ where } L = \text{lit}(\mathcal{L}) \\ &\quad \text{and } A = \text{cond}(\mathcal{L}) \} \\ &\quad \cup \{ \langle \text{pred}(H), \text{pred}(H) \rangle \mid \text{there exists } R \in P : H \in \text{head}(R) \text{ and} \\ &\quad R \text{ is a choice rule} \} \\ \mathfrak{D}_1(P) &= \mathfrak{D}_1^+(P) \cup \mathfrak{D}_1^-(P) \end{aligned}$$

We can draw the one-step dependency relation as a directed graph. For example, the dependency graph of the program in Example 4.1.2 is shown in Figure 4.1.

We now generalize the one-step dependency relation to a full dependency relation. The intuition is that a predicate p depends on a predicate q if there is a path from p to q in the dependency graph. If at least one of the arcs between p and q is negative, then p depends negatively on q .

Definition 4.2.2 *A dependency path π_P of a logic program P is a sequence*

$$\pi_P = \langle p_1, p_2, \dots, p_n \rangle$$

²This will be examined further in Chapter 5.

where $p_i \in \text{Preds}(P)$ for $1 \leq i \leq n$ and $\langle p_j, p_{j+1} \rangle \in \mathfrak{D}_1(P)$ for $1 \leq j < n$. A path π_P is negative (denoted by π_P^-) if and only if $\langle p_j, p_{j+1} \rangle \in \mathfrak{D}_1^-(P)$ for some $1 \leq j < n$. The set of all dependency paths of P is denoted by Π_P and the set of all negative dependency paths of P is denoted by Π_P^- .

Definition 4.2.3 (Dependency Relation) The dependency relation $\mathfrak{D}(P) \subseteq \text{Preds}(P) \times \text{Preds}(P)$ of a logic program P is defined as follows:

$$\mathfrak{D}(P) = \{ \langle p, q \rangle \mid \text{there exists } \pi \in \Pi_P : \pi = \langle p, \dots, q \rangle \} .$$

The negative dependency relation $\mathfrak{D}^-(P) \subseteq \text{Preds}(P) \times \text{Preds}(P)$ of P is defined as follows:

$$\mathfrak{D}^-(P) = \{ \langle p, q \rangle \mid \text{there exists } \pi^- \in \Pi_P^- : \pi^- = \langle p, \dots, q \rangle \} .$$

4.3 OMEGA-STRATIFICATION

Under the usual definition [3] a program is stratified if we can create a stratification for it such that a predicate p that depends negatively on a predicate q is on a higher stratum than q . For example,

$$a \leftarrow \text{not } b$$

is stratified since we can put b on the stratum 0 and a on stratum 1. On the other hand,

$$a \leftarrow \text{not } b$$

$$b \leftarrow \text{not } a$$

is not since a and b depend negatively on each other.

Now we extend the concept by adding a new stratum for the predicates that depend negatively on each other.

Definition 4.3.1 (ω -stratification) An ω -stratification of a CCP P is a function

$$\mathcal{S} : \text{Preds}(P) \rightarrow \mathbb{N} \cup \{\omega\}$$

such that:

1. for each $\langle p_1, p_2 \rangle \in \mathfrak{D}(P)$ it holds that $\mathcal{S}(p_1) \geq \mathcal{S}(p_2)$; and
2. for each $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$ it holds that $\mathcal{S}(p_1) > \mathcal{S}(p_2)$ or $\mathcal{S}(p_1) = \omega$.

We use the convention that $\omega \geq \omega$ and $\omega > n$ for all $n \in \mathbb{N}$. The first condition asserts that a predicate p_1 that depends positively on a predicate p_2 has to be on at least as high a stratum as p_2 . The second condition states that if p_1 depends negatively on p_2 , then p_1 has to be on a higher stratum or they both must be in the ω -stratum.

Definition 4.3.2 Let \mathcal{S} be an ω -stratification of a CCP P . Then, \mathcal{S} is a strict ω -stratification if and only if

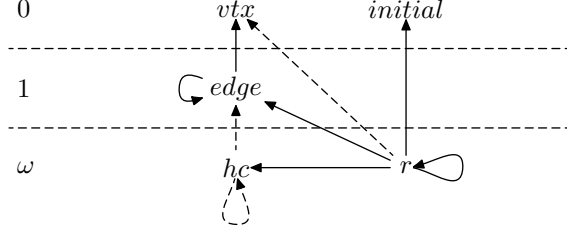


Figure 4.2: A strict ω -stratification of the Hamiltonian cycle program

1. $\mathcal{S}(p_1) > \mathcal{S}(p_2)$ whenever $\langle p_1, p_2 \rangle \in \mathfrak{D}(P)$, $\langle p_2, p_1 \rangle \notin \mathfrak{D}(P)$, and $\mathcal{S}(p_2) < \omega$;
2. for all $p_1 \in \text{Preds}(P)$ it holds that if $\mathcal{S}(p_1) = \omega$, then there exists a predicate $p_2 \in \text{Preds}(P)$ such that $\mathcal{S}(p_2) = \omega$ and $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$.

Intuitively, a stratification is strict when it assigns all dependent predicates that do not necessarily have to be on a same stratum to different strata and does not put any predicate into the ω -stratum if that is not necessary. Later in this section we will present an algorithm that will compute a strict ω -stratification for an arbitrary cardinality constraint program P if we are given its dependency graph.

Example 4.3.1 Consider Example 4.1.2. We can construct a strict ω -stratification \mathcal{S} for the program by looking at its dependency graph. As there are no arcs leading from $vtx/1$ or $initial/1$, we set $\mathcal{S}(vtx) = \mathcal{S}(initial) = 0$. As $edge/2$ depends on vtx , we have $\mathcal{S}(edge) = 1$. As $hc/2$ depends negatively on itself and $r/1$ depends on it, we are forced to set $\mathcal{S}(hc) = \mathcal{S}(r) = \omega$.

Example 4.3.2 Consider the program P :

$$\begin{aligned} \text{odd}(X + 1) &\leftarrow \text{even}(X), \text{number}(X) \\ \text{even}(X + 1) &\leftarrow \text{odd}(X), \text{number}(X) \end{aligned}$$

Here we can set $\mathcal{S}(\text{number}) = 0$ since it depends on nothing. As $\text{odd}/1$ and $\text{even}/1$ depend on each other positively, we set $\mathcal{S}(\text{odd}) = \mathcal{S}(\text{even}) = 1$ to complete the strict stratification \mathcal{S} .

Since the set of predicate symbols $\text{Preds}(P)$ of a proper CCP program is by definition finite, the following proposition immediately follows:

Proposition 4.3.1 *The number of non-empty strata in an ω -stratification of a proper cardinality constraint program is finite.*

4.4 DOMAIN PREDICATES

We divide the predicate symbols into two classes, domain predicates that are on finite strata and non-domain predicates that are on the ω -stratum.

Definition 4.4.1 (Domain Predicate) Let P be a CCP and \mathcal{S} be one of its strict ω -stratifications. Then, $p \in \text{Preds}(P)$ is a domain predicate under \mathcal{S} if and only if $\mathcal{S}(p) < \omega$. The set of all domain predicates of P under \mathcal{S} is denoted by $\mathcal{D}^{\mathcal{S}}(P)$. The set of all non-domain predicates is denoted by $\mathcal{P}^{\mathcal{S}}(P)$.

Definition 4.4.2 (Domain Literal) Let P be a CCP and \mathcal{S} be one of its strict ω -stratifications. A constraint literal $C = \text{Card}(1, \{A : \top\})$ that occurs in the body of a rule $R \in P$ is a domain literal under \mathcal{S} if and only if A is an atom and $\mathcal{S}(\text{pred}(A)) < \mathcal{S}(\text{pred}(H))$ for all atoms $H \in \text{head}(R)$. The set of domain literals that occur in the body of a rule R is denoted by $\text{body}_{\mathcal{D}}^{\mathcal{S}}(R)$.

Note that since there is only one basic literal in a domain literal $C = \text{Card}(1, \{A\})$, we can extend our pred-notation to cover also them so that $\text{pred}(C) = \text{pred}(A)$.

One important thing to notice is that every strict stratification of a program is equivalent in the sense that they all have the same domain predicates and all the rules have the same domain literals under them so we can drop the stratification from our notation and use simply $\mathcal{D}(P)$ and $\text{body}_{\mathcal{D}}(R)$.

Theorem 4.4.1 Let P be a proper CCP and $\mathcal{S}_1, \mathcal{S}_2$ be two of its strict ω -stratifications. Then for all predicate symbols $p \in \text{Preds}(P) : \mathcal{S}_1(p) = \omega$ if and only if $\mathcal{S}_2(p) = \omega$.

Proof. Suppose that there is $p_1 \in \text{Preds}(P)$ such that $\mathcal{S}_1(p_1) = n \in \mathbb{N}$ and $\mathcal{S}_2(p_1) = \omega$ for two strict ω -stratifications of P .

By Definition 4.3.1 every predicate symbol that is placed on the ω -stratum of a strict stratification has to depend negatively on a predicate symbol that is on the same stratum. Thus, there exists a predicate symbol p_2 such that $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$ and $\mathcal{S}_2(p_2) = \omega$. Now p_2 needs a similar dependency. Thus, we have a dependency path:

$$\pi = \langle p_1, p_2, p_3, \dots \rangle \in \Pi_P^-$$

where $\mathcal{S}_2(p_i) = \omega$ for all $i \in \mathbb{N}$. Since the set of predicate symbols $\text{Preds}(P)$ is by definition finite for proper programs, we cannot have an infinite non-looping dependency path in Π_P^- . Thus, π has the form $\langle p_1, p_2, \dots, p_i, \dots, p_i \rangle \in \Pi_P^-$ where at least one predicate symbol p_i depends negatively on itself.

Since $\langle p_i, p_i \rangle \in \mathfrak{D}^-(P)$, Condition 2. of Definition 4.3.1 forces that $\mathcal{S}_1(p_i) = \omega$ so $\mathcal{S}_1(p_1) = \omega$, which contradicts our assumption. \square

Theorem 4.4.2 Let P be a CCP, $\mathcal{S}_1, \mathcal{S}_2$ be two of its strict ω -stratifications, and R be a rule in P . Then,

$$\text{body}_{\mathcal{D}}^{\mathcal{S}_1}(R) = \text{body}_{\mathcal{D}}^{\mathcal{S}_2}(R) .$$

Proof. Let C be a literal such that $C \in \text{body}_{\mathcal{D}}^{\mathcal{S}_1}(R)$ but $C \notin \text{body}_{\mathcal{D}}^{\mathcal{S}_2}(R)$ and H be an atom in the head of R . Then, by Definition 4.2.3,

$$\langle \text{pred}(H), \text{pred}(C) \rangle \in \mathfrak{D} .$$

As C is not a domain literal under \mathcal{S}_2 , we know that $\mathcal{S}_2(\text{pred}(C)) \geq \mathcal{S}_2(\text{pred}(H))$. As $\langle \text{pred}(H), \text{pred}(C) \rangle \in \mathfrak{D}$, Definition 4.3.1 requires that $\mathcal{S}_2(\text{pred}(H)) \geq \mathcal{S}_2(\text{pred}(C))$. Thus, $\mathcal{S}_2(\text{pred}(H)) = \mathcal{S}_2(\text{pred}(C))$.

Suppose that $\mathcal{S}_2(\text{pred}(C)) = \omega$. Then, from Theorem 4.4.1 it follows that $\mathcal{S}_1(\text{pred}(C)) = \omega$ so $C \notin \text{body}_{\mathcal{D}}^{\mathcal{S}_1}(R)$, which contradicts our assumption. Thus, $\mathcal{S}_2(\text{pred}(C)) = n$ for some $n \in \mathbb{N}$.

Since \mathcal{S}_2 is strict and $\mathcal{S}_2(\text{pred}(C)) < \omega$, the condition 1. of Definition 4.3.2 requires that $\langle \text{pred}(C), \text{pred}(H) \rangle \in \mathfrak{D}$. However, this implies that $\mathcal{S}_1(\text{pred}(C)) \geq \mathcal{S}_1(\text{pred}(H))$. Thus, $\mathcal{S}_1(\text{pred}(C)) = \mathcal{S}_1(\text{pred}(H))$ so $C \notin \text{body}_{\mathcal{D}}^{\mathcal{S}_1}(R)$.

Thus, the assumption that $\text{body}_{\mathcal{D}}^{\mathcal{S}_1}(R) \neq \text{body}_{\mathcal{D}}^{\mathcal{S}_2}(R)$ leads to a contradiction. \square

4.5 OMEGA-VALUATION AND RESTRICTION

Next, we will extend the ω -stratification to cover also rules and variables by defining the concept of an ω -valuation.

Definition 4.5.1 (ω -valuation) *The ω -valuation of a rule R under an ω -stratification \mathcal{S} is the function:*

$$\Omega(R, \mathcal{S}) = \min\{\mathcal{S}(\text{pred}(H)) \mid H \in \text{head}(R)\} .$$

The ω -valuation of a global variable V in a rule R under an ω -stratification \mathcal{S} is the function:

$$\Omega(V, R, \mathcal{S}) = \min(\{\mathcal{S}(\text{pred}(A)) \mid A \in \text{body}_{\mathcal{D}}(R) \text{ and } V \in \text{Var}_t(A)\} \cup \{\omega\})$$

Example 4.5.1 *Let \mathcal{S} be as defined in Example 4.3.1. Consider the rule R :*

$$r(Y) \leftarrow 1 \{r(X), \text{initial}(X)\}, hc(X, Y), \text{edge}(X, Y)$$

Now

$$\begin{aligned} \Omega(R, \mathcal{S}) &= \mathcal{S}(r) = \omega \\ \Omega(X, R, \mathcal{S}) &= \min\{\mathcal{S}(\text{edge}), \omega\} = 1 \\ \Omega(Y, R, \mathcal{S}) &= \min\{\mathcal{S}(\text{edge}), \omega\} = 1 . \end{aligned}$$

The atom $\text{initial}(X)$ does not restrict X even though it is a domain predicate since the constraint atom that it occurs in is not a domain literal.

Definition 4.5.2 (ω -restriction) *A conditional literal $X.L : A$ is ω -restricted under a stratification \mathcal{S} if and only if $X \subseteq \text{Var}_t(A)$ and $A = \top$ or $\mathcal{S}(\text{pred}(L)) > \mathcal{S}(\text{pred}(A))$.*

A rule is ω -restricted if all conditional literals in it are restricted and if all global variables that occur in it occur also in a positive body literal that belongs to a strictly lower stratum than the head. A program is ω -restricted if all its rules are ω -restricted.

Definition 4.5.3 A cardinality constraint program P is ω -restricted if and only if there exists a strict stratification \mathcal{S} such that for all rules $r \in P$ it holds that

$$\text{for all } V \in \text{Var}_g(r) : \Omega(V, r, \mathcal{S}) < \Omega(r, \mathcal{S}) .$$

and all conditional literals that occur in P are ω -restricted under \mathcal{S} .

Example 4.5.2 Consider the rule R :

$$s(f(X)) \leftarrow s(X) .$$

This rule is not ω -restricted since for all stratifications \mathcal{S} , $\Omega(R, \mathcal{S}) = \Omega(X, R, \mathcal{S})$.

Next we show that if the program is ω -restricted under one strict ω -stratification, it is restricted under all.

Theorem 4.5.1 Let P be a proper CCP and \mathcal{S}_1 and \mathcal{S}_2 be two of its strict ω -stratifications. Then, P is ω -restricted under \mathcal{S}_1 if and only if it is ω -restricted under \mathcal{S}_2 .

Proof. Suppose that P is ω -restricted under \mathcal{S}_1 but not under \mathcal{S}_2 . Then, there are two possibilities.

1. There exists a conditional literal $\mathcal{L} = X.L : A$ in P where \mathcal{L} is ω -restricted under \mathcal{S}_1 but not under \mathcal{S}_2 . By Definitions 4.2.1 and 4.2.3, we have $\langle \text{pred}(L), \text{pred}(A) \rangle \in \mathfrak{D}^-(P)$. Since \mathcal{L} is restricted under \mathcal{S}_1 , we have $\mathcal{S}_1(\text{pred}(L)) > \mathcal{S}_1(\text{pred}(A))$ so $\mathcal{S}_1(\text{pred}(A)) < \omega$ and A is a domain predicate. Then by Theorem 4.4.1 $\mathcal{S}_2(\text{pred}(A)) < \omega$. Now Definition 4.3.1 requires that $\mathcal{S}_2(\text{pred}(L)) > \mathcal{S}_2(\text{pred}(A))$. This contradicts the assumption that \mathcal{L} is not ω -restricted under \mathcal{S}_2 .
2. There exists some rule $R \in P$ such that R is not ω -restricted under \mathcal{S}_2 . However, by Theorem 4.4.2 the sets of domain literals are the same under both stratifications so the literals that restrict the variables under \mathcal{S}_1 also restrict them under \mathcal{S}_2 and we get another contradiction.

Thus, if P is ω -restricted under one strict ω -stratification, it is restricted under all of them. \square

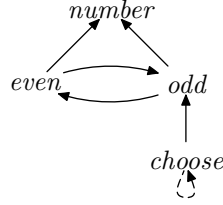
4.6 COMPUTING DOMAIN PREDICATES

While not all CCPs are ω -restricted, it turns out that all of them have strict ω -stratifications. We will give an algorithm that creates a strict ω -stratification of a program P . Since we already proved that all strict stratifications are equivalent³, it is enough that we find one of them.

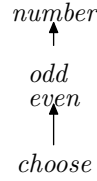
³Theorems 4.4.1–4.5.1.

$$\begin{aligned}
\text{even}(X + 1) &\leftarrow \text{odd}(X), \text{number}(X) \\
\text{odd}(X + 1) &\leftarrow \text{even}(X), \text{number}(X) \\
\{\text{choose}(X)\} &\leftarrow \text{odd}(X)
\end{aligned}$$

(a) Program



(b) Dependency graph



(c) SCC graph

Figure 4.3: An example of SCC graph formation.

The algorithm works by computing the strongly connected components of the dependency graph and then assigning the components to different strata. A strongly connected component of a graph is a maximal subgraph where there exists a path between each pair of nodes in the component. Here we divide the components into two classes, positive and negative. A component is negative if it contains at least one negative arc and all other components are positive.

Definition 4.6.1 *Let P be a cardinality constraint program. Then, its strongly connected component graph $SCC(P) = \langle V_P, E_P, N_P \rangle$ is defined as follows:*

1. $v \in V_P$ if and only if the following three conditions hold:
 - (a) $v \subseteq \text{Preds}(P)$;
 - (b) for all $x, y \in v$ it holds that $\langle x, y \rangle \in \mathfrak{D}(P)$; and
 - (c) if $x \in v$ and there exists $y \in \text{Preds}(P)$ such that $\langle x, y \rangle \in \mathfrak{D}(P)$ and $\langle y, x \rangle \in \mathfrak{D}(P)$, then also $y \in v$.
2. There is an arc $\langle v_1, v_2 \rangle \in E_P$ if and only if there is $x \in v_1$ and $y \in v_2$ such that $\langle x, y \rangle \in E_P$.
3. $N_P \subseteq V_P$ such that

$$N_P = \{v \mid \text{there exists } x, y \in v : \langle x, y \rangle \in \mathfrak{D}^-(P)\} .$$

The nodes of $SCC(P)$ correspond to the strongly connected components of the dependency graph of P . The set N_P contains all those

```

function create-stratification(Program  $P$ )
  Let  $\mathcal{S}$  be an empty stratification
  Let  $G := \langle V, E, N \rangle$  be the SCC graph of  $P$ 
  foreach  $v \in V$  do
    find-stratum( $\langle V, E, N \rangle, v, \mathcal{S}$ )
  end foreach
  return  $\mathcal{S}$ 
end function

function find-stratum(Graph  $\langle V, E, N \rangle$ , Component  $v$ , Stratification  $\mathcal{S}$ )
   $s := 0$ 
  if  $v \in N$  then  $s := \omega$ 
  foreach  $v'$  such that  $\langle v, v' \rangle \in E$  do
     $s' := \text{find-stratum}(\langle V, E, N \rangle, v', \mathcal{S})$ 
    if  $s' \geq s$  then  $s := s' + 1$ 
  end foreach
  foreach  $p \in v$  do
     $\mathcal{S}(p) := s$ 
  end foreach
  return  $s$ 
end function

```

Figure 4.4: An algorithm for creating an ω -stratification

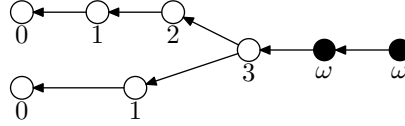


Figure 4.5: A sample SCC graph and its stratification

strongly connected components that contain predicates that depend negatively on itself.

The algorithm *create-stratification* (Figure 4.4) first marks all components in N_P as belonging to the ω -stratum and then computes a depth-first search on $\text{SCC}(P)$. The nodes of V_P that have no successors and do not belong to N_P are allocated on the 0-stratum. If a node v has successors, then we first compute recursively the strata where they belong and take the maximum s of them. Then, we assign every predicate symbol in v to the stratum $s + 1$ where we use the convention that $\omega + 1 = \omega$. In Figure 4.5 we see a sample SCC graph where nodes belonging to N_P are colored black and the resulting stratification that the algorithm computes for it.

We can make this algorithm more efficient by storing the return values of *find-stratum* so that we do not have to compute them again if the computation visits the same component more than once.

Proposition 4.6.1 *Let P be a CCP. Then, the result \mathcal{S} computed by the*

algorithm *create-stratification* is a strict ω -stratification of P .

Proof. Suppose that for predicate symbols p_1 and p_2 it holds that $\langle p_1, p_2 \rangle \in \mathfrak{D}(P)$. Then there are two possibilities. If $\langle p_2, p_1 \rangle \in \mathfrak{D}(P)$, both predicates belong to the same SCC and *find-stratum* assigns the same stratum for both of them, so $\mathcal{S}(p_1) \geq \mathcal{S}(p_2)$. Otherwise, they belong to different components and there is an arc in SCC graph from the component of p_1 to the component of p_2 . As *find-stratum* assigns a component to the stratum $s + 1$ where s is the maximum of the strata of its successors, we see that $\mathcal{S}(p_1) \geq \mathcal{S}(p_2)$ where the equality holds only if $\mathcal{S}(p_2) = \omega$.

Consider the case where $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$. If $\langle p_2, p_1 \rangle \notin \mathfrak{D}(P)$, *find-stratum* behaves as above and $\mathcal{S}(p_1) > \mathcal{S}(p_2)$ or $\mathcal{S}(p_2) = \mathcal{S}(p_1) = \omega$. On the other hand, if $\langle p_2, p_1 \rangle \in \mathfrak{D}(P)$, then both p_1 and p_2 belong to the same component $v \in N_p$ so *find-stratum* assigns both to the ω -stratum. Thus, \mathcal{S} is an ω -stratification.

We see that \mathcal{S} is strict since the only time when dependent predicates p_1 and p_2 are assigned on the same stratum is when they both depend on each other or when they both are on the ω -stratum. Moreover, only those predicates that belong to some $v \in N_p$ or depend on such a predicate are assigned to the ω -stratum so also the other requirement of strictness is met. \square

Corollary 4.6.1 *Every cardinality constraint program has a strict stratification.*

Theorem 4.6.1 *The problem of deciding whether a finite cardinality constraint program is ω -restricted is decidable in polynomial time.*

Proof. The SCC graph of a cardinality constraint program P can be constructed in linear time using, for example, the well-known Tarjan algorithm [172, pp. 481–483]. After that the algorithm *create-stratification* creates a strict ω -stratification \mathcal{S} for P using a quadratic amount of time.⁴ After that, we can check that each rule of P is ω -restricted in a linear time. \square

⁴The algorithm can be modified to achieve a linear time bound by caching results of *find-stratum*.

5 DECIDABILITY OF OMEGA-RESTRICTED PROGRAMS

In this chapter we show that stable models of ω -restricted CCPs are always finite and the question of their existence is decidable. We do so by defining the concept of a *relevant instantiation*. Instead of using the full Herbrand instantiation that we used in defining the models, we use one of its subsets where we can guarantee that the set of stable models of the subset is the same as the corresponding set for the full instantiation.

The key insight to notice is that a Herbrand instantiation can contain infinitely many rules that have unsatisfiable bodies. Those rules cannot generate atoms into the model so they can be left out without affecting the set of stable models. We do not aim to create the smallest possible instantiation for a program because that would be computationally intractable.¹ Instead, we use a compromise where we examine only the positive domain literals in the rule bodies. It turns out that the extensions of domain predicates are the same in every stable model of the program. We discard all those ground instances that contain a domain literal that is not satisfied by this fixed part of the models.

Example 5.0.1 *Consider the ω -restricted program:*

$$\begin{aligned} a(0) &\leftarrow \\ b(f(0)) &\leftarrow \\ c(X) &\leftarrow a(X). \end{aligned}$$

The Herbrand instantiation is infinite since it contains a ground rule:

$$c(f^i(0)) \leftarrow a(f^i(0))$$

for every $i \in \mathbb{N}$. However, only one of them is relevant since $a(0)$ is the only body atom that has a rule for it.

We begin by stating the key theorem:

Theorem 5.0.2 *The problem of existence of a stable model of an ω -restricted CCP is decidable.*

We devote the rest of the chapter for proving this by presenting a method for computing the stable models. Informally, the result follows from the fact that the domain predicates are defined by a subprogram that is stratifiable in the usual sense. We can compute the extensions of domain predicates by examining one stratum at a time. Since no variables are allowed in the 0-stratum, the predicate extensions are all finite. Then, at each new stratum each variable has to occur in a domain literal that belongs to a previous stratum and this keeps the number of ground rules with satisfiable bodies finite. The same thing holds also

¹If we have a rule with just one positive atom in its body, we can leave it out if the literal is false in every answer set of the program. However, the question whether an atom is true in any stable model is **NP**-complete even for normal programs [38], so it is intractable to know whether we can leave it out or not.

for conditional literals: their expansion contains only a finite number of satisfiable literals.

We continue by defining the concepts of *stratum programs* and *domain models*. Then we use *relevant instantiation* to show that the domain models are finite. Finally, we put the pieces together by showing that the stable models of the program coincide with the stable models of the relevant instantiation of the program.

5.1 STRATUM PROGRAMS

We divide the rules that occur in a program into a number of stratum programs. The stable models are constructed in parts so that at each step we instantiate the current stratum program with respect to the atoms that belong to the least model of the previous strata. We will prove that all stratum programs have a unique stable model.

Definition 5.1.1 (Stratum Program) *Let P be an ω -restricted CCP and \mathcal{S} be its strict ω -stratification. Then, the stratum program $P_k^{\mathcal{S}}$ is defined as follows:*

$$P_k^{\mathcal{S}} = \{R \in P \mid \Omega(R, \mathcal{S}) = k\} .$$

A k th partial domain program is a union of first k stratum programs:

$$P_{\leq k}^{\mathcal{S}} = \bigcup_{i \leq k} P_i^{\mathcal{S}} .$$

The domain program $P_{\mathcal{D}}^{\mathcal{S}}$ is the program:

$$P_{\mathcal{D}}^{\mathcal{S}} = \bigcup_{i \in \mathbb{N}} P_i^{\mathcal{S}} .$$

Our next step is to prove that the domain program has a unique stable model that is finite. To prove the uniqueness we introduce the CCP versions of *stratified programs* [27] and *splitting sets* [117].

Splitting Sets

Splitting sets were originally defined by Lifschitz and Turner [117] for normal logic programs. We can divide a program into two parts: top and bottom and then express its stable models as combinations of models of top and bottom programs. Here we generalize the notion to cardinality constraint programs.

We start by introducing an auxiliary notation for turning a set of atoms into a set of facts.

Definition 5.1.2 *Let M be a set of ground atoms. Then, the set $F(M)$ of facts is defined as:*

$$F(M) = \{\langle A, \emptyset \rangle \mid A \in M\} .$$

Example 5.1.1 Let $M = \{a, b, c\}$. Then, $F(A)$ is the program:

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

Definition 5.1.3 Let P be a ground CCP. Then, a set of atoms $U \subseteq \text{Atoms}(P)$ is a splitting set if and only if for all rules $R \in P$:

$$\text{head}(R) \subseteq U \text{ implies that } \text{Atoms}(R) \subseteq U .$$

The set $B_U = \{R \in P \mid \text{head}(R) \subseteq U\}$ is called the bottom of P with respect to U and $T_U = P \setminus B_U$ the top.

A set U is a splitting set of a non-ground CCP P if and only if it is a splitting set of $\text{inst}_{\mathbf{H}}(P)$.

Next, we introduce a notation for creating a modified top program based on a stable model of the bottom.

Definition 5.1.4 Let P be a ground CCP, U its splitting set and M a stable model of the bottom B_U . Then

$$P(U, M) = T_U \cup F(M) .$$

We can take a stable model for the bottom and then substitute it as facts back to the program and obtain a stable model of the top.

Lemma 5.1.1 Let P be a ground CCP, U its splitting set, and M a stable model of B_U . If S is a stable model of $P(U, M)$, then S is a stable model of P .

Proof. Suppose that S is a stable model of $P(U, M)$. First note that $S \cap U = M$ since we added all atoms in M as facts to $P(U, M)$ and there are no rules that could derive other atoms in U in T_U .

Next, by Definition 5.1.3 the bottom B_U does not have any atoms that belong to the top T_U , so its reduct $B_U^S = B_U^M$ so $\mathbf{MM}(B_U^S) = \mathbf{MM}(B_U^M) = M = \mathbf{MM}(F(M)^S)$.

Thus,

$$\begin{aligned} \mathbf{MM}(P^S) &= \mathbf{MM}(B_U^S \cup T_U^S) = \mathbf{MM}(F(M)^S \cup T_U^S) \\ &= \mathbf{MM}((F(M) \cup T_U)^S) = \mathbf{MM}(P(U, M)^S) = S , \end{aligned}$$

so S is a stable model of P . □

Lemma 5.1.2 Let P be a ground CCP and U be its splitting set. If S is a stable model of P , then $S_B = S \cap U$ is a stable model of the bottom B_U and S is a stable model of $P(U, S_B)$.

Proof. Let S be a stable model of P , $S_B = S \cap U$, and $S_T = S \setminus S_B$. Since B_U does not refer to any atoms in the top T_U , we have $B_U^S = B_U^{S_B}$.

Suppose that S_B is not a stable model of B_U . Then either

1. S_B contains an atom that is not in $\mathbf{MM}(B_U^{S_B})$; or

2. $\mathbf{MM}(B_U^{S_B})$ contains an atom that is not in S_B .

Since T_U does not have rules for atoms in U , an atom that is in S_B but not in $\mathbf{MM}(B_U^{S_B})$ is missing also from $\mathbf{MM}(B_U^S)$ and thus also from $\mathbf{MM}(P^S)$ so S is not a stable model of P . In the other case the extra atom in $\mathbf{MM}(B_U^{S_B})$ is present also in $\mathbf{MM}(B_U^S)$ and $\mathbf{MM}(P^S)$ so S is again not a stable model of P . Thus, S_B is a stable model of B_U .

Next,

$$P(U, S_B) = T_U \cup F(S_B)$$

and

$$\begin{aligned} \mathbf{MM}(P(U, S_B)^S) &= \mathbf{MM}(T_U^S \cup F(S_B)) = \mathbf{MM}(T_U^S \cup B_U^S) \\ &= \mathbf{MM}(P^S) = S \end{aligned}$$

so S is a stable model of $P(U, S_B)$. □

These two lemmas lead us to the next corollary:

Corollary 5.1.1 *Let P be a ground CCP and U be its splitting set. Then S is a stable model of P if and only if $S_B = S \cap U$ is a stable model of B_U and S is a stable model of $P(U, S_B)$.*

Stratified Programs

In this section we show that cardinality constraint programs that are stratified in the usual sense [27] have unique stable models. Since we have already defined the concept of ω -stratification, we use it as the building block when defining the standard one: a program is stratified if its ω -stratum is empty.

Definition 5.1.5 *A cardinality constraint program P is stratified if there exists a strict ω -stratification \mathcal{S} where $P_\omega^S = \emptyset$.*

The reason why all stratified CCPs have least models is that we can compute the model from bottom up, starting from the 0-stratum and advancing one stratum at a time. Whenever we have a negation in the program, the extensions of the predicate symbols that occur under it have been already decided. This result holds for all such CCPs, not only for ω -restricted ones.

Theorem 5.1.1 *A stratified CCP P has a unique stable model.*

To prove this we first show that a program that has only one possible reduct has a unique stable model. If every set of atoms generates the same reduct, then the least model of that reduct is necessarily the unique stable model.

Lemma 5.1.3 *Let P be a ground CCP. If for any sets of atoms $M_1, M_2 \in \text{Atoms}(P)$ it holds that $P^{M_1} = P^{M_2}$, then P has a unique stable model.*

Proof. Consider the reduct P^{M_1} of some set of atoms M_1 . By Theorem 3.4.1 there exists the least model $M = \mathbf{MM}(P^{M_1})$. Since we assume

that $P^M = P^{M_1}$, $M = \mathbf{MM}(P^M)$ and M is a stable model of P . It is the only stable model because $P^{M_2} = P^M$ for every set of atoms M_2 . \square

Now we can prove Theorem 5.1.1

Proof.[Of Theorem 5.1.1] Let \mathcal{S} be an ω -stratification of P that fulfills the condition of Definition 5.1.5. Next, we prove by induction over the strata that P has a unique stable model. We do it by showing that all stratum programs have unique reducts and using Lemma 5.1.3.

First, consider the 0-stratum. By Definition 4.3.1 we know that the stratum program $P_0^{\mathcal{S}}$ cannot contain any negative literals (neither basic nor cardinality) and all conditional literals in it have to be of the form $A : \top$ where A is an atom. Additionally, all rules in it are basic rules.

A reduct removes all negative literals from the program. Since they cannot occur in the 0-stratum, we get the same reduct $\text{inst}_{\mathbf{H}}(P_0^{\mathcal{S}})$ no matter what set of atoms M we use to create it. Thus, by Lemma 5.1.3 it has a unique stable model M_0 .

Next, suppose that there exists some $k \in \mathbb{N}$ such that $P_{\leq k}^{\mathcal{S}}$ has a unique stable model M_k .

Consider the program $P_{\leq k+1}^{\mathcal{S}}$. The set of atoms:

$$U = \{\text{head}(R) \mid R \in P_{\leq k}\}$$

is a splitting set of $P_{\leq k+1}^{\mathcal{S}}$ where:

$$\begin{aligned} B &= P_{\leq k}^{\mathcal{S}} \\ T &= P_{k+1}^{\mathcal{S}} . \end{aligned}$$

The set M_k is the only stable model of B . By Corollary 5.1.1 any stable model of $P_{\leq k+1}^{\mathcal{S}}$ has to agree with M_k with respect to the set U . That is, if M_{k+1} is a stable model of $P_{\leq k+1}^{\mathcal{S}}$, then

$$\begin{aligned} M_k &\subseteq M_{k+1} \text{ and} \\ M_{k+1} \cap (U \setminus M_k) &= \emptyset . \end{aligned}$$

Consider a set of atoms $M' \subseteq \text{Atoms}(P_{\leq k+1}^{\mathcal{S}})$ where $M_k \subseteq M'$. When we examine the reduct of $P_{\leq k+1}^{\mathcal{S}}$ with respect to M' , we note that:

1. All predicate symbols occurring in a negative cardinality literal not $\text{Card}(b, S)$ belong to previous strata and $M' \models \text{not Card}(b, S)$ exactly when $M_k \models \text{not Card}(b, S)$.
2. All negative literals occurring in a rule also belong to an earlier stratum and so $\text{Card}(b, S)^{M'} = \text{Card}(b, S)^{M_k}$ for all cardinality atoms and stable models M' of $P(T, M_k)$.

Thus, the reduct $\text{inst}_{\mathbf{H}}(P(T, M_k))^{M'}$ is the same for any set M' and by Lemma 5.1.3 $P_{\leq k+1}^{\mathcal{S}}$ has a unique stable model. \square

We use the opportunity to also introduce the concept of *local stratification*. We examine the dependencies at the level of instantiated atoms instead of predicate symbols. A program is locally stratified if its Herbrand instantiation is stratified. For the purpose of this definition we will use the notation $\mathfrak{D}_l(P)$ to denote the dependency relation $\mathfrak{D}_l(P) \subseteq$

$\text{Atoms}(\text{inst}_{\mathbf{H}}(P)) \times \text{Atoms}(\text{inst}_{\mathbf{H}}(P))$ that is defined analogously to Definition 4.2.3 but over the rules of $\text{inst}_{\mathbf{H}}(P)$ instead of P , and \mathcal{S}_l to denote an ω -stratification based on $\mathfrak{D}_l(P)$.

Definition 5.1.6 *A CCP P is locally stratified if there exists a strict ω -stratification \mathcal{S} of $\text{inst}_{\mathbf{H}}(P)$ such that $\text{inst}_{\mathbf{H}}(P)_{\omega}^{\mathcal{S}} = \emptyset$.*

Proposition 5.1.1 *A locally stratified CCP P has a unique stable model.*

Proof. We can create a stratified CCP P' that is equivalent to $\text{inst}_{\mathbf{H}}(P)$ by replacing all atoms $p(t_1, \dots, t_n)$ in it by new 0-ary predicate symbol $P_{p(t_1, \dots, t_n)}$. Essentially, we move the arguments of the predicate into its name. As P' is stratified, it has a unique stable model. \square

5.2 UNIQUENESS OF DOMAIN PROGRAM

Even though an ω -restricted CCP may have an infinite number of different ω -stratifications, they all are essentially the same and it does not matter which one we choose to use since the domain program is the same under all of them.

Proposition 5.2.1 *Let P be an ω -restricted CCP. Then, it has a unique domain program $P_{\mathcal{D}}$ and a unique ω -program P_{ω} .*

Proof. This follows directly from Theorem 4.4.1 that states that the set of domain predicates is the same under every stratification. \square

Since the domain program is exactly the stratifiable part of a CCP, we immediately get the result that it has a unique stable model.

Theorem 5.2.1 *Let P be an ω -restricted CCP. Then, the domain program $P_{\mathcal{D}}$ has a unique stable model.*

Proof. By Proposition 5.2.1 the domain program $P_{\mathcal{D}}$ does not depend on the strict ω -stratification \mathcal{S} that we choose to use. Since $P_{\mathcal{D}}$ contains exactly those rules that belong to the finite strata under \mathcal{S} , it is stratified in the usual sense. By Theorem 5.1.1 it has a unique stable model. \square

5.3 RELEVANT INSTANTIATION

In a relevant instantiation we are given a set of atoms that define the truth values for domain literals. When we create the instantiation we include only those ground instances where all domain literals are satisfied by the set.

Definition 5.3.1 *Let P be a CCP, U a universe, and M a set of atoms. Then, the relevant expansion $\mathcal{E}_r(\mathcal{L}, U, M)$ of a conditional literal $\mathcal{L} = X.L : A$ is the set:*

$$\mathcal{E}_r(\mathcal{L}, U, M) = \{ L\sigma : A\sigma \mid \sigma \in \text{subs}(X, U) \text{ and } \exists \sigma' \in \text{subs}(\text{Var}(\mathcal{L}) \setminus X, U_{\mathbf{H}}(P)) : M \models A\sigma\sigma' \}$$

and the relevant expansion of a cardinality constraint $C = \text{Card}(b, S)$ is the constraint:

$$\mathcal{E}_r(C, U, M) = \text{Card}(b, \{\mathcal{E}_r(\mathcal{L}, U, M) \mid \mathcal{L} \in S\}) .$$

Example 5.3.1 Let $U = \{1, 2, 3\}$, $M = \{d(1, 1), d(2, 2)\}$, and $C = \text{Card}(1, \{X.a(X, Y) : d(X, Y)\})$. Then, the relevant expansion of C is:

$$\mathcal{E}_r(C, U, M) = \text{Card}(1, \{a(1, Y) : d(1, Y), a(2, Y) : d(2, Y)\}) .$$

The third possibility, $a(3, Y) : d(3, Y)$ is not included since there is no atom of the form $d(3, y)$ in M .

Definition 5.3.2 The relevant instantiation of a rule $R = \langle H, \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \rangle$ with respect to a set of atoms M and a universe U is the set of rules:

$$\text{inst}_{\mathbf{Hr}}(R, U, M) = \{ \langle H\sigma, \{\mathcal{E}_r(\mathcal{C}_1, U, M)\sigma, \dots, \mathcal{E}_r(\mathcal{C}_n, U, M)\sigma\} \rangle \mid \sigma \in \text{subs}(\text{Var}(P), U) \text{ and } M \models \text{body}_{\mathcal{D}}(R\sigma) \}$$

where H is either A or $\{A_1, \dots, A_n\}$ where A and A_i are atoms.

Definition 5.3.3 The relevant Herbrand instantiation of a program P is the set of rules:

$$\text{inst}_{\mathbf{Hr}}(P, M) = \bigcup_{R \in P} \text{inst}_{\mathbf{Hr}}(R, U_{\mathbf{H}}(P), M) .$$

Note that if the set M is finite, then also the relevant instantiation is finite. We will be using this result later so we formally proof it.

Lemma 5.3.1 Let $R = H \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$ be rule in a CCP P and M be a finite set of atoms. Then, $\text{inst}_{\mathbf{Hr}}(R, U_{\mathbf{H}}(P), M)$ is finite.

Proof. There are two possible sources of infinity in $\text{inst}_{\mathbf{Hr}}(R, U_{\mathbf{H}}(P), M)$:

1. a conditional literal may have an infinite expansion; and
2. there may be an infinite number of relevant ground instances of R .

First, note that for a ground conditional literal $L' : A'$ to belong to the expansion of a conditional literal $X.L : A$, there have to be two substitutions σ and σ' such that $L' = L\sigma$ and $A\sigma\sigma' \in M$. Since M is finite, it contains only a finite number of ground instances of A so the number of suitable substitutions is also finite.

Next, all literals in $\text{body}_{\mathcal{D}}(R\sigma)$ are ground literals of the form $\mathcal{L} = \text{Card}(1, \{A : \top\})$ so we see that $M \models \mathcal{L}$ only when $A \in M$. Thus, only a finite number of ground instances may satisfy all domain literals. \square

5.4 DOMAIN MODELS

Next we define the concept of domain models and partial domain models. The $(k + 1)$ th partial domain model is the stable model of the relevant instantiation of the k th partial domain program, and the domain model is the union of all partial domain models. We will later prove that the domain model is the unique stable model of the domain program.

Definition 5.4.1 (Domain Model) *Let P be an ω -restricted CCP and \mathcal{S} be its strict ω -stratification. Then, the partial domain models $D_k^{\mathcal{S}}$ are defined as follows:*

$$\begin{aligned} D_0^{\mathcal{S}} &= \mathbf{MM}(P_0^{\mathcal{S}}) \\ D_{k+1}^{\mathcal{S}} &= \mathbf{MM}(\text{inst}_{\mathbf{Hr}}(P_{k+1}^{\mathcal{S}}, D_k^{\mathcal{S}}) \cup F(D_k^{\mathcal{S}})) . \end{aligned}$$

The domain model $D_{\omega}^{\mathcal{S}}$ is defined as

$$D_{\omega}^{\mathcal{S}} = \bigcup_{i \in \mathbb{N}} D_i^{\mathcal{S}} .$$

Note that since all partial domain programs are stratified, they have unique stable models.

5.5 FINITENESS OF DOMAIN MODELS

We will now show that all partial domain models of finite programs are finite. Since there is only a finite number of non-empty strata, there are only a finite number of partial domain models in the union that forms the domain model so it is also finite.

Theorem 5.5.1 *Let P be a finite CCP that is ω -restricted under the strict ω -stratification \mathcal{S} . Then, for all $i \in \mathbb{N}$ the partial domain model $D_i^{\mathcal{S}}$ is finite.*

Proof. We do an induction over the strata of \mathcal{S} . In the basic case $i = 0$ we have $D_0^{\mathcal{S}} = \mathbf{MM}(P_0^{\mathcal{S}})$ that is finite. Suppose that the claim holds for some $k \geq 0$, that is, $D_k^{\mathcal{S}}$ is finite. Then, $D_{k+1}^{\mathcal{S}} = \mathbf{MM}(\text{inst}_{\mathbf{Hr}}(P_{k+1}^{\mathcal{S}}, D_k^{\mathcal{S}}) \cup F(D_k^{\mathcal{S}}))$. As our induction hypothesis is that $D_k^{\mathcal{S}}$ is finite, we can apply Lemma 5.3.1 to see that $D_{k+1}^{\mathcal{S}}$ is finite. \square

Theorem 5.5.2 *Let P be a finite CCP that is ω -restricted under the strict ω -stratification \mathcal{S} . Then, its domain model $D_{\omega}^{\mathcal{S}}$ is finite.*

Proof. This theorem follows directly from Theorem 5.5.1 and the fact that \mathcal{S} may have only a finite number of nonempty strata since all finite programs are proper. \square

5.6 STABLE MODELS OF OMEGA-RESTRICTED PROGRAMS

In this section we prove that the stable models of a finite ω -restricted program P coincide with the stable models of the relevant instantiation of P_ω .

We do this in two parts. First, we show that using relevant instantiation instead of the full instantiation for the domain program does not lose any atoms from the domain model and then we extend it to cover also the ω -program.

Theorem 5.6.1 *Let P be a CCP that is ω -restricted under the ω -stratification \mathcal{S} . Then, the domain model $D_\omega^\mathcal{S}$ is the unique stable model of the domain program $P_\mathcal{D}$.*

Proof. By Theorem 5.2.1 $P_\mathcal{D}$ has a unique stable model. Suppose that $\mathbf{MM}(\text{inst}_\mathbf{H}(P_\mathcal{D})) \neq D_\omega^\mathcal{S}$.

For the 0-stratum we have that $\text{inst}_\mathbf{H}(P_0) = \text{inst}_{\mathbf{Hr}}(P_0, \emptyset) = P_0$ since all rules at that stratum have to be ground. Thus, $\mathbf{MM}(\text{inst}_\mathbf{H}(P_0)) = \mathbf{MM}(\text{inst}_{\mathbf{Hr}}(P_0, \emptyset))$.

For the induction hypothesis suppose that there exists some k such that for all strata $i \leq k$ it holds that $\mathbf{MM}(P_{\leq i}) = D_i^\mathcal{S}$.

Next, suppose that $\mathbf{MM}(P_{\leq k+1}) \neq D_{k+1}^\mathcal{S}$. Then, there exists an atom A over which the two models disagree. By induction hypothesis $D_k^\mathcal{S} = \mathbf{MM}(\text{inst}_\mathbf{H}(\mathbf{MM}(P_{\leq k})))$, so

$$D_{k+1}^\mathcal{S} = \mathbf{MM}(\text{inst}_{\mathbf{Hr}}(P_{k+1}, D_k^\mathcal{S}) \cup F(D_k^\mathcal{S})) \subseteq \mathbf{MM}(P_{\leq k+1})$$

since $\text{inst}_{\mathbf{Hr}}(P_{k+1}, D_k^\mathcal{S}) \subseteq \text{inst}_\mathbf{H}(P_i)$. Thus, A is true in $\mathbf{MM}(P_{\leq k+1})$ but false in $D_{k+1}^\mathcal{S}$. Furthermore, we choose A so that it has the shortest possible derivation in $\mathbf{MM}(P_{\leq k+1})$. Then, there has to be a rule

$$A \leftarrow \text{body}$$

in $\text{inst}_\mathbf{H}(P_{\leq k+1})$ where body is satisfied in $\mathbf{MM}(\text{inst}_\mathbf{H}(P_{\leq k+1}))$ but not in $D_{k+1}^\mathcal{S}$. This implies that there exists a cardinality literal \mathcal{C} in body such that $\mathbf{MM}(\text{inst}_\mathbf{H}(P_{\leq k+1})) \models \mathcal{C}$ but $D_{k+1}^\mathcal{S} \not\models \mathcal{C}$.

This means further that \mathcal{C} contains at least one atom B over which the two models disagree. Since we assumed that the models agree up to the k -stratum, we know that $\mathcal{S}(\text{pred}(B)) = k + 1$. There are now two possibilities:

1. $B \in \mathbf{MM}(\text{inst}_\mathbf{H}(P_{\leq k+1}))$ but $B \notin D_{k+1}^\mathcal{S}$. In this case B has a shorter derivation in $\text{inst}_\mathbf{H}(P_{\leq k+1})$ than A , which contradicts our assumption that A has the shortest derivation.
2. $B \in D_{k+1}^\mathcal{S}$ but $B \notin \mathbf{MM}(\text{inst}_\mathbf{H}(P_{\leq k+1}))$. This is not possible since we earlier showed that $D_{k+1}^\mathcal{S} \subseteq \mathbf{MM}(\text{inst}_\mathbf{H}(P_{\leq k+1}))$.

As it is not possible to find an atom A over which the two models disagree, we conclude that

$$\mathbf{MM}(P_{\leq k+1}) = \mathbf{MM}(\text{inst}_{\mathbf{Hr}}(P_{k+1}, D_k^\mathcal{S}) \cup F(D_k^\mathcal{S})) = D_{k+1}^\mathcal{S}.$$

Thus, $\mathbf{MM}(\text{inst}_{\mathbf{H}}(P_{\mathcal{D}})) = D_{\omega}^{\mathcal{S}}$. □

From this result we immediately get the following corollary:

Corollary 5.6.1 *An ω -restricted CCP has a unique domain model.*

Since the domain model is unique, we will drop the stratification \mathcal{S} from it and write only D_{ω} .

5.6.1 The Complete Stable Models

The main result of this section is that the set of stable models of a program stays the same if we use relevant instantiation instead of the complete one. We show this via splitting sets.

Theorem 5.6.2 *Let P be an ω -restricted CCP. Then,*

$$\mathbf{SM}(\text{inst}_{\mathbf{H}}(P)) = \mathbf{SM}(\text{inst}_{\mathbf{Hr}}(P_{\omega}, D_{\omega}) \cup F(D_{\omega})).$$

Proof. The definition of ω -restriction² requires that no predicate symbol that occurs in the domain program $P_{\mathcal{D}}$ appears as a head in P_{ω} . Thus, the set:

$$D = \bigcup_{R \in \text{inst}_{\mathbf{H}}(P_{\mathcal{D}})} \text{head}(R)$$

is a splitting set of $P' = \text{inst}_{\mathbf{H}}(P)$. Next, by Theorem 5.6.1 D_{ω} is the unique stable model of $P_{\mathcal{D}}$. By Corollary 5.1.1 all stable models of P are stable models of $P'(U, D_{\omega})$.

Consider a rule R that does not belong to the relevant instantiation. The relevant instantiation was defined so that all rules that are left out from it contain at least one unsatisfied domain literal. Thus, $D_{\omega} \not\models \text{body}_{\mathcal{D}}(R)$, so R is trivially satisfied by all model candidates. This also means that R cannot be used to justify any atom in the model, so its existence does not alter the set of stable models in any way and it may be dropped.

Thus, the stable models of P are determined by $\text{inst}_{\mathbf{Hr}}(P_{\omega}, D_{\omega}) \cup F(D_{\omega})$. □

5.7 PUTTING IT ALL TOGETHER

Now we have all bits and pieces that we need to prove Theorem 5.0.2.

Theorem 5.0.2 *The problem of existence of a stable model of an ω -restricted CCP is decidable.*

Proof. The algorithm in Figure 5.1 presents a naive and inefficient algorithm for testing the existence of a stable model.

²Definition 4.5.3.

```

function has-stable-model(Program  $P$ )
   $D := \emptyset$ 
   $\mathcal{S} := \text{create-stratification}(P)$ 
   $i := 0$ 
  while  $P_i$  is not empty do
     $PG := PG \cup \text{naive-instantiate-relevant}(P_i, D)$ 
     $D := D \cup \text{MM}(PG)$ 
     $i := i + 1$ 
  endwhile
   $PG := PG \cup \text{naive-instantiate-relevant}(P_\omega, D)$ 
  foreach  $M \in 2^{\text{Atoms}(PG)}$  do
    if  $\text{MM}(PG^M) = M$  then
      return true
    endif
  endfor
  return false
endfunction

function naive-instantiate-relevant(Ruleset  $R$ , Atomset  $D$ )
   $S := F(D)$ 
  foreach  $r \in R$  do
     $S' := \{\text{inst}_{\mathbf{H}_r}(r', U_{\mathbf{H}}(P), D) \mid r' \in \text{inst}(r, D) \text{ and } D \models \text{body}_D(r')\}$ 
     $S := S \cup S'$ 
  end foreach
  return  $S$ 
end function

```

Figure 5.1: A naive algorithm for testing the existence of a stable model of an ω -restricted CCPs.

We go through the nonempty finite strata of the program and at each step we create the relevant instantiation of the current stratum and compute its least model. By Proposition 4.3.1 there is only a finite number of them.

By Lemma 5.3.1 each relevant instantiation is finite and we can create them by examining every non-ground rule belonging to a stratum one-by-one and checking through the previously-computed partial domain models to see what variable substitutions are possible.

When we have instantiated the current stratum, we compute its stable model. Since the instantiation is finite, we can do this, for example by explicitly testing every possible subset of the atoms. Finally, we instantiate the rules of the P_ω with respect to the domain model D_ω and systematically go through all subsets of the atoms occurring in the program to see if one of them is a stable model. By Theorem 5.6.2 a stable model of the relevant instantiation $\text{inst}_{\mathbf{H}_r}(P_\omega, D_\omega) \cup F(D_\omega)$ is a stable model of $\text{inst}_{\mathbf{H}}(P)$. \square

5.8 INSTANTIATION AS A DATABASE OPERATION

We can interpret the domain predicates of an ω -restricted program as relations on its Herbrand universe. With this interpretation the stratum programs correspond closely to Datalog⁻ programs. The reason why we are interested in this interpretation is that we can use standard database techniques in creating the relevant instantiation in a more efficient manner than the naive algorithms presented in previous sections.

We start by giving a very brief overview of the relational database model. We do not go deep into details and include the minimum that is necessary for our purposes.

Relational Database Model

In *relational database model* [202] we think a database as a set of relations and all items occur as tuples of the relations. In the database context we usually think that every component of a tuple has a *attribute name* attributed to it. We will take the approach that we use variables as the names and we write them as the arguments for the sets that occur in the relations. For example, if the attribute names for the relation $\mathbf{R}/2$ are X and Y , we write it as $\mathbf{R}[X, Y]$.

With this approach we can think that a two-ary predicate symbol $p/2$ is represented by some relation with two attributes X and Y where the X corresponds to the first argument of $p(X, Y)$ and Y to the second.

The four fundamental database operations that we are interested in are *projection*, *renaming*, *selection*, and *joining*.

Projection

A projection is a function that removes some components from the tuples of the relation. For example, if we have a relation

$$\mathbf{R}[X, Y, Z] \subseteq \mathbf{A}[X] \times \mathbf{B}[Y] \times \mathbf{C}[Z] ,$$

then the projection $\pi_{X,Y}$ defines the function:

$$\pi_{X,Y}(\mathbf{R}) = \{ \langle x, y \rangle \mid \text{there exists } z : \langle x, y, z \rangle \} .$$

In this work we use the set semantics for projection. This means that we get only one tuple $\langle x, y \rangle$ into the projection even if there are more than one possible value for z .

Renaming

In renaming we change some of the attributes names of a relation. A renaming $\rho_{X/Y}$ changes the name of the attribute X to Y but keeps otherwise the relation intact.

Selection

A selection is an operation that allows us to pick a subset of tuples of the relation. Formally, we define some predicate P that is then evaluated over the tuples. The result is the set of tuples that satisfy the predicate. Formally, a selection σ_P defines the function:

$$\sigma_P(\mathbf{R}) = \{ t \mid t \in \mathbf{R} \text{ and } P(t) \text{ is true } \} .$$

Natural Join

We often want to combine two or more relations into one. The most common way how we do is to use the natural join. The natural join $\mathbf{R} \bowtie \mathbf{S}$ of relations \mathbf{R} and \mathbf{S} is simplest to describe as the result of the following algorithm:

1. compute the Cartesian product $\mathbf{R} \times \mathbf{S}$;
2. from the product select the tuples where all the attributes that belong to both relations have identical values; and
3. project away the second copy of duplicated attributes.

Formally, a natural join can be defined using a selection and projection operators.

Example 5.8.1 *Suppose that we have two relations, $\mathbf{R}[X, Y]$ and $\mathbf{S}[Y, Z]$ that are defined as follows:*

$$\begin{aligned}\mathbf{R}[X, Y] &= \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\} \\ \mathbf{S}[Y, Z] &= \{\langle 2, 1 \rangle, \langle 2, 3 \rangle, \langle 1, 4 \rangle\} .\end{aligned}$$

Then, their natural join $\mathbf{T}[X, Y, Z] = \mathbf{R}[X, Y] \bowtie \mathbf{S}[Y, Z]$ is the ternary relation:

$$\mathbf{T}[X, Y, Z] = \{\langle 1, 2, 1 \rangle, \langle 1, 2, 3 \rangle\} .$$

Aggregates

An aggregate is a function that computes a value based on a set of tuples. Most aggregates are relatively simple arithmetic operations computing sums, averages, and other useful statistics.

Conceptually aggregates are often included in the database relations as special attributes whose values are computed based on other values. The advantage of this approach is that it allows us to make selections based on the aggregate.

Example 5.8.2 *Consider again the relation $\mathbf{R}[X, Y]$ from Example 5.8.1. If we want find out how many values occur in the second place of the relations, we can do it with the expression:*

$$\mathbf{count}(\pi_Y(R)) = \mathbf{count}(\{\langle 2 \rangle, \langle 3 \rangle\}) = \{\langle 2 \rangle\}$$

*where **count** is an aggregate that returns a unary relation that contains only one tuple whose value tells how many different tuples there were. This version of **count** illustrates the idea of including aggregates in relation attributes.*

In this work we consider only simple aggregates that are monotonic and not recursively defined. We can always evaluate these kinds of aggregates and get a unique relation as a result [202].

Datalog

The Datalog [26] language is essentially a subset of Prolog that was designed as a logic programming interface to relational databases. Plain Datalog programs are composed using positive normal rules and the predicate symbols are divided into two classes: *data* and *program* predicates.³ Data predicates are defined by an existing relational database and may not occur in the heads of rules, and program predicates define new relations based on data predicates.

The semantics of plain Datalog is such that each program predicate symbol p/n defines an n -ary relation \mathbf{R}_p where $\langle a_1, \dots, a_n \rangle \in \mathbf{R}_p$ exactly when $p(a_1, \dots, a_n)$ is true in the least model of the program.

Datalog⁻ [27] adds range-restricted stratified negation to the language. Since stratified programs have a least model, the semantics stays unchanged.

There is a well-known naive algorithm (Figure 5.2) to compute the semantics of a Datalog program directly [202]. The algorithm is essentially the same as that we use to compute the least model of a positive simple program: we start by initializing the relations corresponding to program predicates with the empty set and then iteratively generate new tuples to the relations until we reach a fixpoint.

The main workhorse of the algorithm is the function

$$eval\text{-}rule(R, \mathbf{I}_1, \dots, \mathbf{I}_k, \mathbf{Q}_1, \dots, \mathbf{Q}_k)$$

that takes as its argument a rule R and two sets of relations, \mathbf{I}_i for data predicates and \mathbf{Q}_i for program predicates, and it then creates an expression of relational algebra for the program predicate that occurs in the head of the rule. The expression is essentially a large natural join over the relations of the body, and then we project it to the variables that occur in the head.

Example 5.8.3 Suppose that we have the rule:⁴

$$p(X, Y, Z) \leftarrow d_1(X, Y), d_2(Y, Z)$$

the body corresponds to the natural join $\mathbf{R}_{d_1} \bowtie \mathbf{R}_{d_2}$ of the relations corresponding to the data predicates d_1 and d_2 , so we set $\mathbf{R}_p = \mathbf{R}_{d_1} \bowtie \mathbf{R}_{d_2}$.

If a program is non-recursive, we can create the relational algebra expression for every rule and evaluate them in correct order to find the least model of the Datalog program. However, when we have recursively defined predicates in our program, we cannot compute the joins directly.

The naive algorithm solves this problem by constructing the relation iteratively. It starts with the assumption that the relations are empty, and then in each step it adds those tuples in the relation that occur in the heads of rules whose bodies are satisfied, in a manner similar to the T_P one-step provability operator. This process is then continued until we find the least fixed point.

³In the context of database systems data and program predicates are often called *extensional* and *intensional* predicates, respectively.

⁴Here we use p_i for the program predicate symbols and d_i for the data predicates.

```

function naive-datalog(Datalog Program  $P$ )
  foreach  $p \in \text{Preds}(P)$  do
     $\mathbf{R}_p := \emptyset$ 
  endfor
  repeat
    foreach  $p \in \text{Preds}(P)$  do
       $\mathbf{Q}_p := \mathbf{R}_p$ 
    endfor
    foreach  $p \in \text{Preds}(P)$  do
       $\mathbf{R}_p := \text{eval}_P(p, \mathbf{I}_1, \dots, \mathbf{I}_k, \mathbf{Q}_1, \dots, \mathbf{Q}_k)$ 
    endfor
  until  $\mathbf{R}_p = \mathbf{Q}_p$  for all  $p \in \text{Preds}(P)$ 
  return  $\{\mathbf{R}_p \mid p \in \text{Preds}(P)\}$ 
endfunction

function evalP(Predicate  $p$ , Relations  $\mathbf{I}_i, \mathbf{Q}_i$ )
   $\mathbf{R}_p := \emptyset$ 
  foreach  $R \in P|p$  do
     $\mathbf{R}_p := \mathbf{R}_p \cup \text{eval-rule}(R, \mathbf{I}_1, \dots, \mathbf{I}_k, \mathbf{Q}_1, \dots, \mathbf{Q}_k)$ 
  endforeach
  return  $\mathbf{R}_p$ 
endfunction

function eval-rule(Rule  $R$ , Relations  $\mathbf{I}_i, \mathbf{Q}_i$ )
  Create a relational algebra expression  $E$  that corresponds to  $R$ 
  Evaluate  $E$  with respect to relations  $\mathbf{I}_i$  and  $\mathbf{Q}_i$  and return it
endfunction

```

Figure 5.2: A naive algorithm for computing the Datalog semantics

Example 5.8.4 Consider the Datalog program that computes the transitive closure of a binary relation d :

$$p(X, Y) \leftarrow d(X, Y) \quad (1)$$

$$p(X, Z) \leftarrow p(X, Y), p(Y, Z) . \quad (2)$$

Suppose that $\mathbf{R}_d = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle\}$. Since (1) depends on purely data predicates, $\text{eval-rule}(1)$ returns always the same relation:

$$\mathbf{R}_1 = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle\} .$$

During the first iteration for (2) we start by initializing the auxiliary relation $\mathbf{Q}_p^1 = \emptyset$, so $\mathbf{R}_2^1 = \emptyset$ and $\mathbf{R}_p^1 = \mathbf{R}_1 \cup \mathbf{R}_2^1 = \mathbf{R}_d$.

In the next iteration we set $\mathbf{Q}_p^2 = \mathbf{R}_p^1$. Now, the join $\mathbf{Q}_p^2 \bowtie \mathbf{Q}_p^2$ is:

$$\mathbf{Q}_p^2 \bowtie \mathbf{Q}_p^2 = \{\langle 1, 2, 3 \rangle, \langle 2, 3, 4 \rangle\}$$

and when we project it to the variables of the head we get the relation:

$$\mathbf{R}_2^2 = \{\langle 1, 3 \rangle, \langle 2, 4 \rangle\}$$

so we set:

$$\mathbf{R}_p^2 = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle\} .$$

In the next iteration we find that

$$\mathbf{R}_2^3 = \{\langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 4 \rangle\}$$

and

$$\mathbf{R}_p^3 = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 4 \rangle\} .$$

Now we have reached the fixpoint and the algorithm terminates.

The performance of this naive algorithm can be improved in many ways. For example, it is not necessary to consider the whole relation \mathbf{R}_q at each stage and it is enough to just examine the tuples that were added to it in the previous stage. This approach gives us the so-called semi-naive algorithm [202] for Datalog evaluation. Many other improvements have been proposed [203].

When we have range-restricted stratified negation in a rule body, we can express it as set difference of relations [202]. This means that we can use the naive algorithm to evaluate also Datalog^\neg .

Relations and Cardinality Constraint Programs

We can extend the relational approach to CCPs by substituting a different *eval-rule* algorithm into the naive algorithm. The main problem is that it is not possible to express cardinality constraints directly with relational algebra when we use variables and conditional literals.⁵ We can escape this problem by taking the use of the standard **count** aggregate. We have to also handle the conditional literals suitably.

We consider here only positive programs but we can add the negations in the same way as in Datalog^\neg .

⁵In the ground case we can replace a cardinality atom with a corresponding normal logic program construction.

Rules

We create the relation that corresponds to the body of a CCP rule R using the following steps:

1. find a relation $\mathbf{R}_{R,\mathcal{D}}$ for domain literals of the rule;
2. construct the relations $\mathbf{R}_{\mathcal{L}}$ for every conditional literal \mathcal{L} in the rule;
3. select from $\mathbf{R}_{R,\mathcal{D}}$ all tuples that make all cardinality atoms in the rule true.

We need the three steps because of conditional literals and cardinality atoms. We want a set of tuples that correspond to those variable bindings that satisfy the body of the rule. The relation corresponding to the domain literals give us the set of all possible instances that the rule can have, but the rule body may have some additional cardinality atoms in it. We have to check each tuple in $\mathbf{R}_{R,\mathcal{D}}$ to see whether it makes all the remaining cardinality atoms are true or not.

The cardinality atoms may contain conditional literals that have to be expanded. The most convenient way to represent a conditional literal is to make a relation out of it that contains one tuple for every atom that belongs to the expansion⁶. We combine the relations corresponding to conditional literals occurring in a cardinality atom $C = \text{Card}(b, \{\mathcal{L}_1, \dots, \mathcal{L}_n\})$ together as a join $\mathbf{R}_C = \mathbf{R}_{\mathcal{L}_1} \bowtie \dots \bowtie \mathbf{R}_{\mathcal{L}_n}$. We then create a selection of $\mathbf{R}_{R,\mathcal{D}}$ where we choose those tuples whose join with \mathbf{R}_C contains at least b tuples.

Domain Predicate Symbols

We define an n -ary relation \mathbf{R}_p for every n -ary domain predicate symbol in the same way as we have relations for program predicates in Datalog. A tuple $\langle t_1, \dots, t_n \rangle \in \mathbf{R}_p$ exactly when $p(t_1, \dots, t_n)$ is true in the domain model.

Domain Literals

Every global variable that occurs in an ω -restricted rule has to occur also in some domain literal in the rule body. We can take the natural join over the relations of domain literals to get the set of all possible global variable substitutions, exactly the same way as we can create the relation for the body of a Datalog program. We use $\mathbf{R}_{R,\mathcal{D}}$ to denote this join.

Conditional Literals

A conditional literal $\mathcal{L} = X.p(X) : q(X)$ denotes the set of atoms $\{p(t) \mid q(t) \text{ is true}\}$. This gives us the relational algebra expression for \mathcal{L} : we create the natural join of the two relations \mathbf{R}_p and \mathbf{R}_q , and then project it back to the variables of p . We also rename all the local variables so that we do not get confused when a variable occurs as a local variable in more than one literal.

⁶Remember that we are currently considering only positive programs so there are no negative literals.

Formally, if $\mathcal{L} = \{Y_1, \dots, Y_k\}.p(X_1, \dots, X_n) : q(X'_1, \dots, X'_m)$ is a conditional literal, the relational algebra expression corresponding to it is:

$$\mathbf{R}_{\mathcal{L}} = \rho_{Y_1/Z_1, \dots, Y_n/Z_n}(\pi_{X_1, \dots, X_n}(\mathbf{R}_p \bowtie \mathbf{R}_q)) \ .$$

Cardinality Atoms

A cardinality atom $C = \text{Card}(b, \{\mathcal{L}_1, \dots, \mathcal{L}_n\})$ corresponds to a selection: we create the join of the relations $\mathbf{R}_{\mathcal{L}_i}$, and then use the count aggregate to define the selection.

The relation corresponding to the set of conditional literals is:

$$\mathbf{R}_C = \mathbf{R}_{\mathcal{L}_1} \bowtie \dots \bowtie \mathbf{R}_{\mathcal{L}_n}$$

We define the selection predicate as:

$$P_C(t) = b \leq \text{count}(\{t\} \bowtie \mathbf{R}_C) \ .$$

We create a different join for every tuple so that we do not mix together conditional literals corresponding to different global variable bindings.

Finishing Touches

Let R be the rule

$$H \leftarrow C_1, \dots, C_n, D_1, \dots, D_k \ .$$

where $\text{body}_{\mathcal{D}}(R) = \{D_1, \dots, D_k\}$. Then, the relation \mathbf{R}_R is:

$$\mathbf{R}_R = \sigma_{P_{C_1} \wedge \dots \wedge P_{C_n}}(\mathbf{R}_{R, \mathcal{D}}) \ .$$

Example 5.8.5 *We examine a program that has no recursion so that we can see what the complete relations corresponding to the different predicates without having to use the naive algorithm.*

$$\begin{aligned} a(1) &\leftarrow \\ a(2) &\leftarrow \\ b(1) &\leftarrow \\ b(3) &\leftarrow \\ c(X, Y) &\leftarrow a(X), a(Y) \\ d(X) &\leftarrow 2 \{Y.c(X, Y) : a(Y)\}, b(X) \end{aligned}$$

In this case the relations corresponding to $a/1$, $b/1$, and $c/2$ correspond to the Datalog ones:

$$\begin{aligned} \mathbf{R}_a &= \{\langle 1 \rangle, \langle 2 \rangle\} \\ \mathbf{R}_b &= \{\langle 1 \rangle, \langle 3 \rangle\} \\ \mathbf{R}_c &= \{\langle 1, 1 \rangle, \langle 2, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 2 \rangle\} \ . \end{aligned}$$

In the case of d , we have to examine the cardinality atom and the conditional literal in it. For the conditional literal we get:

$$\mathbf{R}_{\mathcal{L}} = \rho_{Y/Z}(\pi_{X,Y}(\mathbf{R}_c[X, Y] \bowtie \mathbf{R}_a[Y])) = \rho_{Y/Z}(\{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle\}) \ .$$

From this we get the following selection predicate for the cardinality atom:

$$P_C(t) = 2 \leq \mathbf{count}(\{t\} \bowtie \mathbf{R}_{\mathcal{L}}) .$$

Now the relation

$$\mathbf{R}_d = \sigma_{P_C}(\mathbf{R}_{R,\mathcal{D}}) = \sigma_{P_C}(\mathbf{R}_b) .$$

Since \mathbf{R}_b has two tuples, we have to do the test in the selection two times. With the tuple $\langle 1 \rangle$ we get

$$P_C(\langle 1 \rangle) = 2 \leq \mathbf{count}(\{\langle 1, 1 \rangle, \langle 1, 2 \rangle\}) = T .$$

Conversely, with $\langle 3 \rangle$ we get:

$$P_C(\langle 3 \rangle) = 2 \leq \mathbf{count}(\emptyset) = F .$$

Thus, the relation \mathbf{R}_d is:

$$\mathbf{R}_d = \{\langle 1 \rangle\}$$

that corresponds to the only global variable binding $X/1$ that satisfies the body of the rule.

Relations and Relevant Instantiation

We conclude this section by showing that the relation $\mathbf{R}_{R,\mathcal{D}}$ corresponds directly with the set of relevant ground instances of the rule. A ground instance $R\sigma$ belongs in the relevant instantiation if $A\sigma$ is true for every domain literal A in $body_{\mathcal{D}}(R)$. We now show that every tuple in $\mathbf{R}_{R,\mathcal{D}}$ corresponds directly to a substitution that satisfies this condition.

In this proof we examine the complete domain model D_{ω} and the complete relations \mathbf{R}_p of domain predicates. We will show later in Chapter 8 that the correspondence holds also when we examine the domain program one stratum at the time.

Proposition 5.8.1 *Let P be an ω -restricted CCP, and $R \in P$ be a rule. Then, $\langle a_1, \dots, a_n \rangle \in \mathbf{R}_{R,\mathcal{D}}[X_1, \dots, X_n]$ if and only if there exists a substitution $\sigma \in \text{subs}(\{X_1, \dots, X_n\}, U_{\mathbf{H}}(P))$ such that*

1. $\sigma(X_i) = c_i$ for all $1 \leq i \leq n$; and
2. $D_{\omega} \models A\sigma$ for every $A \in body_{\mathcal{D}}(R)$.

Proof. First note that we can trivially convert a tuple from $\mathbf{R}_{R,\mathcal{D}}$ into a substitution σ and vice versa by defining that $\sigma(X_i) = a_i$.

Suppose that $\langle a_1, \dots, a_n \rangle \in \mathbf{R}_{R,\mathcal{D}}[X_1, \dots, X_n]$. Consider a domain literal $p(c'_1, \dots, c'_m) \in body_{\mathcal{D}}(R)$. Since the extensions of domain literals are unique (Theorem 5.2.1) and $\mathbf{R}_{R,\mathcal{D}}$ was formed as a natural join over the extensions of the domain literals, $D_{\omega} \models p(c'_1, \dots, c'_m)$.

Next, suppose that there is some substitution σ where $D_{\omega} \models A\sigma$ for every $A \in body_{\mathcal{D}}(R)$. Then, the extensions of the domain predicates contain the ground atoms $A\sigma$, so their natural join $\mathbf{R}_{R,\mathcal{D}}$ also contains a corresponding tuple. \square

Table 6.1: Computational complexity

Variables	Functions	INSTANTIATION	MODEL
No	—	—	NP -complete
Fixed	No	P -complete	NP -complete
	Yes	2- EXP -complete	2- NEXP -complete
Unlimited	No	EXP -complete	NEXP -complete
	Yes	2- EXP -complete	2- NEXP -complete

6 COMPUTATIONAL COMPLEXITY

In this chapter we examine the computational complexity of ω -restricted programs. The results are based on [193, 191] where most of these results were originally proved. We are mainly interested in two different problems:

PROBLEM 1: INSTANTIATION. Given an ω -restricted CCP P and a ground atom $p(t_1, \dots, t_n)$ where p is a domain predicate, does $D_\omega \models p(t_1, \dots, t_n)$ hold?

PROBLEM 2: MODEL. Does an ω -restricted CPP P have a stable model?

Intuitively, INSTANTIATION tells us how difficult it is to create the relevant instantiation of a program with variables and MODEL tells how hard it is to find a stable model.

Throughout this chapter we use only the Herbrand interpretation for function symbols. We leave out interpreted function symbols since we may use arbitrarily complex functions and their evaluation can dominate the instantiation process. Thus, we limit ourselves to the simplest possible case where function evaluation does not significantly affect the necessary time to solve INSTANTIATION or MODEL.

In addition to proving complexity results for the whole class of ω -restricted programs, we examine how the computational complexity of INSTANTIATION and MODEL changes when we restrict our attention to some subclasses of programs. We use two parameters to divide the ω -restricted programs into four classes:

- A rule may contain either an unlimited number of variables or there is some constant limit d for their number.
- The programs can contain arbitrary function symbols or only 0-ary constants.

We use the convention that if the number of variables is fixed, each rule may contain d global variable occurrences and each conditional literal may have d local variable occurrences. We do this to simplify the proofs.

The main complexity results are presented in Table 6.1. The MODEL complexity for ground weight constraint programs with the stable model semantics has been presented in [175] and for normal logic programs in [131, 38]. Since ω -restricted CCPs are essentially a subclass of the more general weight constraint rules, the hardness results of [175] apply.

6.1 BASICS OF COMPUTATIONAL COMPLEXITY

The field of computational complexity examines how difficult different problems are to solve. In general, we are interested in asymptotic complexity: how does the effort to solve the problem increase when the size of the problem instance grows. In this section we present a brief introduction to the complexity theory. A comprehensive introduction can be found, for example, in *Computational Complexity* by Christos Papadimitriou [152].

We use Turing machines [199] as our underlying form of computation. A Turing machine has a control unit that can be in different states and a working tape that acts as a memory during the computation. We will present a formal definition in Section 6.3 when we examine how they can be simulated with CCPs.

If we have a Turing machine that solves some problem, we can examine how much resources it uses in solving different problem instances. The two most common measurements are *time* and *space*. The time complexity tells how many steps the computation lasts and the space complexity tells many working tape cells are used.

A Turing machine can be *time-bounded* by some function $f(n)$. If the input is n symbols long, the machine always halts and gives its answer after using at most $f(n)$ computation steps to do it. Similarly, a *space-bounded* Turing machine always halts and does not use more than $f(n)$ tape cells during the computation.

For example, a problem is in the class of *polynomial time* (**P**TIME or shorter **P**) if there exists some Turing machine M and a $k \geq 0$ such that for every input x , the machine M solves the problem and takes at most $|x|^k$ steps to do it.

A problem that has only two possible answers, YES and NO, is a *decision problem*. When we work with decision problems it is useful to interpret a problem P as a set that contains all problem instances for which the answer is YES. Thus, the notation $x \in P$ denotes that the answer to x is YES and $x \notin P$ denotes that the answer is NO. This notation allows us to define the concept of reduction in a simple way. In particular, $\langle P, A \rangle \in \text{INSTANTIATION}$ denotes that the ground atom A is in the stable model of P_D and $P \in \text{MODEL}$ denotes that P has a stable model.

If we want a more complex answer, we have a *function problem*. For example, the problem of finding a stable model M of P is a function problem. If a decision problem belongs in a complexity class **C**, then the corresponding function problem is in the class **FC**.

Definition 6.1.1 A reduction from a problem P into a problem P' is a function $f : P \rightarrow P'$ such that:

$$x \in P \text{ if and only if } f(x) \in P' .$$

The intuition is that we take an instance of a problem P and solve it by transforming it into an instance of P' and solving that, instead. In

P	Polynomial time (n^k)
NP	Nondeterministic polynomial time
EXP	Exponential time (2^{n^k})
NEXP	Nondeterministic exponential time
2-EXP	Doubly exponential time ($2^{2^{n^k}}$)
2-NEXP	Nondeterministic doubly exponential time

Figure 6.1: Complexity classes

particular, we use polynomial reductions where f has the restriction that we need to be able to compute f in polynomial time.¹

Definition 6.1.2 (Completeness) *A problem P is hard with respect to some complexity class C if all problems in C can be polynomially reduced to P .*

A problem P is complete with respect to some complexity class C if it is both C -hard and in C .

The final concept that we need is the notion of *nondeterministic* complexity classes. A nondeterministic Turing machine may have more than one possible action to execute in each time step so it can have more than one computation for any given input. The Turing machine accepts the input (answers “YES”) if at least one of those computations is accepting.

The complexity classes that we need in this section are shown in Figure 6.1. The class **P** contains all those problems that can be solved in polynomial time using deterministic Turing machines and **NP** extends that to nondeterministic Turing machines. Currently the most important open problem in complexity theory is whether they are the same class or not. The classes **EXP** and **NEXP** contain the problems that can be solved in an exponential time, and **2-EXP** and **2-NEXP** contain the problems that are doubly exponential, that is, their time usage is bounded by some function of the form $2^{2^{n^k}}$.

In Chapter Section 9 we present CCPs to solve some problems that belong to different complexity classes. We will introduce those classes when we first encounter them.

6.2 RELATIONSHIP OF MODEL AND INSTANTIATION

We start the complexity analysis by noting that MODEL is always at least as difficult as INSTANTIATION since for any program P and ground atom A we can create a program $m(P)$ such that $m(P)$ has a stable model only if $D_\omega \models A$.

Definition 6.2.1 *The function $m : \text{INSTANTIATION} \rightarrow \text{MODEL}$ is defined as follows:*

$$m(P) = P_D \cup \{\leftarrow \text{not } A\} .$$

¹When working with some simpler complexity classes we may need to use different forms of reductions, but for this work polynomial time reductions are sufficient.

Proposition 6.2.1 *The function m is a polynomial-time reduction from INSTANTIATION to MODEL.*

Proof. The algorithm from the proof of Theorem 4.6.1 gives us a way to compute $m(P)$ in polynomial time, so we have only to prove that the function preserves solutions.

By Theorem 5.6.1 we know that $P_{\mathcal{D}}$ always has a unique stable model. Thus, the crucial rule is the one that we added to it:

$$\leftarrow \text{not } A .$$

This constraint rejects model candidates where A is not true. Since there is only one candidate model, D_{ω} , the program $m(P)$ has a stable model exactly when $D_{\omega} \models A$. Thus, m is a reduction from INSTANTIATION to MODEL. \square

6.3 TURING MACHINE TRANSLATION

We establish the complexity hardness results in this work by proving that the computations of a Turing machine can be simulated by a logic program. The configurations of the machine are encoded as sets of atoms and a stable model of the program corresponds to one computation, and the atoms that are true in it tell what configurations were visited during the computation.

Since ω -restricted programs are decidable, we cannot simulate an arbitrary Turing machine since as problems that relate to Turing machine computations are undecidable. This limitation arises from the fact that all relevant instantiations are finite. A Turing machine can use an unlimited amount of its work tape so a stable model that captures such a computation would need to be infinite.

However, when we work with a bounded Turing machine, we know that there is an upper bound on the amount of resources that the machine can use. If we give an input x to a Turing machine that is time-bounded by a function $f(n)$, we know that any computation takes at most $f(|x|)$ steps to complete and it can use at most the first $f(|x|)$ cells of the tape since the tape head can move only one cell at a time. This means that it is enough that our relevant instantiation contains rules for the first $f(|x|)$ steps and tape cells.

The Turing machine encoding consists of two parts, a fixed one that implements the transition function of a bounded Turing machine M when its transition relation is given as a set of facts, and an input-specific part that encodes the input x of M as well as its working tape.

The crux of the hardness proofs is to show that we have a polynomial reduction from a time-bounded Turing machine M and an input x to a CCP $TM(M, |x|)$. The fixed part of the encoding needs only ten rules so it can be constructed in constant time. What we need then is a way of creating $f(|x|)$ tape cells and time steps using a polynomial amount of rules. Since we can code the cells and steps as numbers, this amounts to finding a way to express the first $f(|x|)$ natural numbers with a polynomial number of rules.

In addition of having the encoding $TM(M)$ for deterministic Turing machines, we will also present an encoding $NTM(M)$ for nondeterministic machines.

Definition 6.3.1 (Turing Machine) A deterministic Turing machine is a quadruple $M = \langle K, \Sigma, \delta, q_0 \rangle$ where K is a finite set of states, Σ is a finite alphabet containing the blank symbol \sqcup , $q_0 \in K$ is the initial state and δ is a transition function $\delta : K \times \Sigma \rightarrow (K \cup \{y, n\}) \times \Sigma \times \{l, s, r\}$.

Definition 6.3.2 A configuration of a Turing machine $M = \langle K, \Sigma, \delta, s \rangle$ is a quadruple $c = \langle q, \alpha, \sigma, \beta \rangle$ where $q \in K$, $\sigma \in \Sigma$, and $\alpha, \beta \in \Sigma^*$.²

A configuration $c = \langle q, \alpha, \sigma, \beta \rangle$ yields the configuration $c' = \langle q', \alpha', \sigma', \beta' \rangle$ in one step (denoted $c \vdash_M c'$) if and only if one of the following conditions hold:

1. $\beta' = \sigma''\beta$, $\alpha = \alpha'\sigma'$, and $\delta(q, \sigma) = \langle q', \sigma'', l \rangle$;
2. $\beta = \sigma'\beta'$, $\alpha' = \alpha\sigma''$, and $\delta(q, \sigma) = \langle q', \sigma'', r \rangle$; or
3. $\beta' = \beta$, $\alpha' = \alpha$, and $\delta(q, \sigma) = \langle q', \sigma', s \rangle$.

A computation of M with an input $x \in \Sigma^*$ (denoted by $M(x)$) is a sequence of configurations $\langle c_0, c_1, c_2, \dots \rangle$ where the initial configuration $c_0 = \langle q_0, \varepsilon, \sqcup, x \rangle$ and such that for all $i \in \mathbb{N}$ it holds that $c_i \vdash_M c_{i+1}$.

When we run a Turing machine with a given input, there are three possible results:

1. the machine halts in the state y ;
2. the machine halts in the state n ; and
3. the machine goes into an infinite loop and never terminates.

We will be using Turing machines whose time use is bounded so the third possibility cannot happen. If the machine ends up in the state y , we say that it accepts the input, and if in n , it rejects it.

Definition 6.3.3 A Turing machine $M = \langle K, \Sigma, \delta, s \rangle$ terminates on an input x if and only if $M(x)$ is a finite sequence $\langle c_0, c_1, \dots, c_m \rangle$ where c_m does not have any valid successor configuration.

A Turing machine M is time-bounded by a function $f(n)$ if for all $x \in \Sigma^*$, the computation $M(x)$ terminates in at most $f(|x|)$ steps.

A Turing machine M accepts an input x if and only if $M(x)$ terminates and the final configuration c_m is of the form $c_m = \langle y, \alpha, \sigma, \beta \rangle$ where $\sigma \in \Sigma$ and $\alpha, \beta \in \Sigma^*$. The set of all strings accepted by M is denoted with $L(M)$.

A nondeterministic Turing machine is otherwise equivalent to a deterministic one except that instead of having a transition function it has a transition relation that can contain more than one possible next transition for any given configuration and it accepts if at least one computation accepts.

²We may denote a configuration $\langle q, \alpha, \sigma, \beta \rangle$ with a simpler notation $\langle q, \alpha\sigma\beta \rangle$.

Definition 6.3.4 A nondeterministic Turing machine is a quadruple $M = \langle K, \Sigma, \Delta, q_0 \rangle$ where K is a finite set of states, Σ is a finite alphabet containing the blank symbol \sqcup , $q_0 \in K$ is the initial state and Δ is a transition relation $\Delta \subseteq K \times \Sigma \times (K \cup \{y, n\}) \times \Sigma \times \{l, s, r\}$.

We can use the deterministic definitions of configurations and computations also for the nondeterministic case, but we have to change the definition of one computational step.

Definition 6.3.5 Let $M = \langle K, \Sigma, \Delta, s \rangle$ be a nondeterministic Turing machine. Then, a configuration $c = \langle q, \alpha, \sigma, \beta \rangle$ yields the configuration $c' = \langle q', \alpha', \sigma', \beta' \rangle$ if and only if there exists a transition $\langle q, \sigma, q', \sigma'', d \rangle \in \Delta$ such that one of the following conditions hold:

1. $\beta' = \sigma''\beta$, $\alpha = \alpha'\sigma'$, and $d = l$;
2. $\beta = \sigma'\beta'$, $\alpha' = \alpha\sigma''$, and $d = r$; or
3. $\beta' = \beta$, $\alpha' = \alpha$, $\sigma' = \sigma''$, and $d = s$.

Finally, we formalize the nondeterministic acceptance condition.

Definition 6.3.6 A nondeterministic Turing machine $M = \langle K, \Sigma, \Delta, s \rangle$ accepts an input $x \in \Sigma^*$ if and only if there is a computation $\langle c_0, \dots, c_m \rangle$ where c_0 is the initial configuration $\langle s, \sqcup x \rangle$, c_m is of the form $\langle y, \alpha, \sigma, \beta \rangle$ where $\sigma \in \Sigma$ and $\alpha, \beta \in \Sigma^*$ and $c_i \vdash_M c_{i+1}$ for all $0 \leq i \leq m$.

6.4 THE TURING MACHINE ENCODING

We will now examine how we can represent Turing machines as logic programs. We first present the encoding for deterministic machines and then modify it to get a representation of nondeterministic machines.

6.4.1 The Rules of the Encoding

Predicate Symbols

We encode the states of a Turing machine M using the predicate symbol $state(q)$, the alphabet using $symbol(\sigma)$, and the transitions using $transition(q_1, \sigma_1, q_2, \sigma_2, d)$, where $d \in \{l, s, r\}$ denoting left, stationary, and right, respectively.

The tape cells are defined using the predicate $cell/1$ and the time steps with $time/1$. An atom $at-cell(\sigma, c, t)$ denotes that the tape cell c contains the symbol σ at the time step t . An atom $reads(c, t)$ denotes that the tape head is positioned over the cell c at the step t . Finally, an atom $in-state(q, t)$ denotes that the machine is in state q at step t .

Computation Steps

We start by identifying which transition we take at each computation step. For this we use an auxiliary predicate $take(Q_1, S_1, Q_2, S_2, D, T)$ to denote that we take the transition $\langle Q_1, S_1, Q_2, S_2, D \rangle$ at the time step T .

$$\begin{aligned}
take(Q_1, S_1, Q_2, S_2, D, T) \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\
& in\text{-}state(Q_1, T), reads(C, T), \\
& at\text{-}cell(S_1, C, T), time(T), cell(C) .
\end{aligned} \tag{6.1}$$

Once we know which transition to take, we use three rules to execute the computation step. First, we write the correct symbol to the current cell:

$$\begin{aligned}
at\text{-}cell(S_2, C, T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\
& take(Q_1, S_1, Q_2, S_2, D, T), \\
& reads(C, T), next(T, T'), cell(C) .
\end{aligned}$$

Then, we change to the new state:

$$\begin{aligned}
in\text{-}state(Q_2, T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\
& take(Q_1, S_1, Q_2, S_2, D, T), next(T, T')
\end{aligned}$$

Finally, we move the tape head to the correct direction:

$$\begin{aligned}
reads(C', T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\
& take(Q_1, S_1, Q_2, S_2, D, T), reads(C, T), \\
& next\text{-}cell(C, D, C'), next(T, T') .
\end{aligned}$$

The rules above handle the cell where the tape head is currently positioned. In addition, we have to assert that the state of the other tape cells stays constant:

$$\begin{aligned}
at\text{-}cell(S, C, T') \leftarrow & at\text{-}cell(S, C, T), \text{not } reads(C, T), \\
& cell(C), next(T, T'), symbol(S) .
\end{aligned}$$

Tape Cell Adjacency Relation

The predicate *next-cell/3* connects the adjacent tape cells together so that the tape head can be moved to the correct direction:

$$\begin{aligned}
next\text{-}cell(C_1, r, C_2) \leftarrow & next(C_1, C_2) \\
next\text{-}cell(C_1, l, C_2) \leftarrow & next(C_2, C_1) \\
next\text{-}cell(C, s, C) \leftarrow & cell(C) .
\end{aligned} \tag{6.2}$$

Halting States

Finally, we want to recognize whether the Turing machine halts in an accepting state or not:

$$\begin{aligned}
accept \leftarrow & in\text{-}state(y, T), time(T) \\
reject \leftarrow & in\text{-}state(n, T), time(T) .
\end{aligned} \tag{6.3}$$

Numerals

We do not give here the rules for the predicates *time/1*, *cell/1*, *next/2* that encode the time steps, tape cells, and the successor relation. Instead, in the following complexity proofs we show how we can define them with a polynomial number of rules using tools that are available for the four different ω -restricted program classes.

The encodings will be based on numerals. That is, we show how we can create numbers from 0 to $f(n)$ where $f(n)$ is the upper bound for computation length and then use them for identifying time steps and tape cells.

Initial Configuration

When a Turing machine starts its operation with an input $x = x_1x_2 \cdots x_n$, the symbols of the input are positioned in the first n positions of the tape and all other cells are empty.

We encode this using the rules:

$$\begin{aligned} at-cell(x_1, t_1, t_0) &\leftarrow \\ &\vdots \\ at-cell(x_n, t_n, t_0) &\leftarrow \\ part-of-input(t_1) &\leftarrow \\ &\vdots \\ part-of-input(t_n) &\leftarrow \end{aligned}$$

where the terms t_i are representations of first $n + 1$ numerals.

All tape cells that are not part of the input are empty:

$$at-cell(\sqcup, C, t_0) \leftarrow cell(C), \text{ not } part-of-input(C) .$$

The Turing machine starts from the initial state and the tape head is over the first tape cell, reading the empty symbol that precedes the actual input:

$$in-state(q_0, t_0, x_1, t_1) \leftarrow .$$

Nondeterministic Turing Machines

We can generalize the translation to allow nondeterministic Turing machines by forcing the machine to choose between possible transitions at all computation steps. We can do this by changing the rule for *take/6* to choose exactly one of the possible transitions:

$$\begin{aligned} \{take(Q_1, S_1, Q_2, S_2, D, T)\} &\leftarrow transition(Q_1, S_1, Q_2, S_2, D), \\ &in-state(Q_1, T), reads(C, T), \\ &at-cell(S_1, C, T), time(T), cell(C) \\ &\leftarrow 2 \{ \{Q_1, S_1, Q_2, S_2, D\}.take(Q_1, S_1, Q_2, S_2, D, T) : \\ &transition(Q_1, S_1, Q_2, S_2, D)\}, \\ &time(T) \\ &\leftarrow \text{not } 1 \{ \{Q_1, S_1, Q_2, S_2, D\}.take(Q_1, S_1, Q_2, S_2, D, T) : \\ &transition(Q_1, S_1, Q_2, S_2, D)\}, \\ &time(T) \end{aligned}$$

All other rules in $NTM(M)$ are equal to the corresponding rules of $TM(M)$.

A similar translation for normal logic programs has been presented earlier by V. W. Marek and J. B. Remmel in [127].

6.4.2 Correctness of Turing Machine Simulation

In this section we argue that the Turing machine translation that we presented is correct. We will choose one possible representation for the numerals and then show that the least model of the translation corresponds exactly to the computation of a bounded Turing machine.

Completing the Translation

We examine a version of the Turing machine simulation program where we encode the cell positions with integers. We will call that machine with the name $TM(M, n)$ where n is the bound on the number of computation steps for the machine M . In this case the rules that correspond to the initial configuration for a given input $x = x_1x_2 \cdots x_m$ will be:

$$\begin{aligned} at-cell(x_1, 1, 0) &\leftarrow \\ part-of-input(1) &\leftarrow \\ at-cell(x_2, 2, 0) &\leftarrow \\ part-of-input(2) &\leftarrow \\ &\vdots \\ at-cell(x_m, m, 0) &\leftarrow \\ part-of-input(m) &\leftarrow . \end{aligned}$$

The definitions for $time/1$, $cell/1$, and $next/2$ will be based on the bound of the computation time:

$$\begin{aligned} number(1) &\leftarrow \\ &\vdots \\ number(n) &\leftarrow \\ next(0, 1) &\leftarrow \\ &\vdots \\ next(n-1, n) &\leftarrow \\ time(X) &\leftarrow number(X) \\ cell(X) &\leftarrow number(X) . \end{aligned}$$

At the beginning we are in the initial state q_0 and look at the empty symbol preceding the actual input:

$$\begin{aligned} in-state(q_0, 0) &\leftarrow \\ reads(0, 0) &\leftarrow \end{aligned}$$

Even though the formal definition of Turing machines implies a computation tape that extends infinitely to one direction, we need only a finite subset of the tape since the number of cells that a bounded Turing machine is restricted by the bound.

Uniqueness of Stable Models

First we show that $TM(M, n)$ has only one stable model. We do it by showing that it is locally stratified.

Proposition 6.4.1 *Let M be a Turing machine. Then, the program $TM(M, n)$ has a unique stable model.*

Proof. By Proposition 5.1.1 a locally stratified CCP has a unique stable model. We now show that there are no negative cycles among atoms in $\text{inst}_{\mathbf{H}}(TM(M, n))$.

Only two rules have negative literals in them. The first is:

$$\begin{aligned} at-cell(S, C, T') \leftarrow & at-cell(S, C, T), \text{not } reads(C, T), \\ & cell(C), next(T, T'), symbol(S) \end{aligned}$$

The dependency graph of $TM(M, n)$ contains a negative cycle:

$$at-cell \rightarrow reads \rightarrow take \rightarrow at-cell$$

so we see that $TM(M, n)$ is not stratified. When we examine the heads of the rules that participate in the cycle, we see that the cycle has the form:

$$\begin{aligned} at-cell(S, C, T + 2) \rightarrow & reads(C, T + 1) \rightarrow take(Q_1, S_1, Q_2, S_2, D, T) \\ \rightarrow & at-cell(S, C, T) \end{aligned}$$

When we instantiate this, we get a number of dependency paths of the form:

$$\begin{aligned} at-cell(s, p, n) \rightarrow & reads(p, n - 1) \rightarrow take(q_1, s_1, q_2, s_2, d, n - 2) \\ \rightarrow & at-cell(s, p, n - 2) \rightarrow reads(p, n - 3) \rightarrow \dots \\ \rightarrow & at-cell(s, p, 2) \rightarrow reads(p, 1) \rightarrow take(q_1, s_1, q_2, s_2, d, 0) \\ \rightarrow & at-cell(s, p, 0) . \end{aligned}$$

As the sequences have no cycles, they are locally stratified.

The other rule with a negative literal is:

$$at-cell(\sqcup, C, 0) \leftarrow cell(C), \text{not } part-of-input(C) .$$

Since $part-of-input/1$ is defined using only facts, this rule causes no cycles in the dependency graph.

As no ground atom depends negatively on itself, $TM(M, n)$ is locally stratified. Thus, it has a unique stable model. \square

This result does not hold for the translation for nondeterministic Turing machines. Such machines can have more than one computation on a given input so the corresponding translation has more than one stable model and it cannot be locally stratified.

Deterministic Simulation

We will now complete the argument that the stable model of $TM(M)$ corresponds to the computation of M .

Proposition 6.4.2 *Let $M = \langle K, \Sigma, \delta, q_0 \rangle$ be a deterministic Turing machine that is time-bounded by a function $f(n)$ and $x \in \Sigma^*$ be its input. Then, $x \in L(M)$ if and only if *accept* is true in the stable model of $TM(M, f(|x|))$.*

Proof. The program $TM(M, n)$ has a unique stable model by Proposition 6.4.1 and we define it with C .³ Let $C(k)$ denote the subset of C :

$$\begin{aligned} C(k) = & \{at-cell(\sigma, i, k) \mid at-cell(\sigma, i, k) \in C\} \\ & \cup \{in-state(s, k) \mid in-state(s, k) \in C\} \\ & \cup \{reads(c, k) \mid reads(c, k) \in C\} . \end{aligned}$$

Next we show that the atoms in each set $C(k)$ correspond to the configuration $c_k = \langle s, \alpha, \sigma, \beta \rangle$ of M at the computation step k . That is, there is exactly one atom $at-cell(\sigma, i, k)$ for each i and σ is the same symbol that occurs in the i th tape cell in c_k . Also, there is exactly one atom $in-state(s, k)$ and one atom $reads(p, k)$ where s and p are the same as the state and the position of the tape head of M at c_k .

Consider first $k = 0$. Since the zero does not have any predecessors in the *next/2* relation, the only rules for atoms in $C(0)$ are those that were defined as facts and with the rule:

$$at-cell(\sqcup, C, 0) \leftarrow cell(C), \text{ not } part-of-input(C) .$$

The atoms $at-cell(\sigma, i, 0)$ where $i \in [1, |x|]$ correspond to the input symbols x_i and for all other tape cells we have an atom $at-cell(\sqcup, i, 0) \in C(0)$. Similarly, the only atoms for *in-state/2* and *reads/2* were defined by facts that set the state to the initial state s and tape head position to the cell 0. Thus, $C(0)$ corresponds to the initial configuration $\langle s, \sqcup x \rangle$.

Next, suppose that for some $k \geq 0$ it holds that $C(k)$ corresponds to the configuration c_k . We now consider the set $C(k+1)$.

Consider the predicate *take/6*. It has only one rule in $TM(M, n)$, namely:

$$\begin{aligned} take(Q_1, S_1, Q_2, S_2, D, T) \leftarrow & transition(Q_1, S_1, Q_2, S_2, D), \\ & in-state(Q_1, T), reads(C, T), \\ & at-cell(S_1, C, T), time(T), cell(C) . \end{aligned} \quad (6.4)$$

By induction hypothesis the set $C(k)$ corresponds to the configuration c_k . This means that there is exactly one atom $in-state(q, k)$, one $reads(p, k)$, and one $at-cell(s, p, k)$ in it and they correspond to the state, tape head position, and the symbol that is under the tape head in c_k . As M is deterministic, there is exactly one fact $\langle transition(q, s, q', s', d), \emptyset \rangle$ in $TM(M, n)$. When we examine the ground instances of (6.4) where $T = k$, we see that exactly one of them has a satisfied body in $C(k)$ so we can derive the atom $take(q, s, q', s', d, k)$ into C , and that atom corresponds to the transition δ_k that M took at c_k .

³Here we depart from our usual convention of denoting models with M because we need that symbol to denote the Turing machine.

The ground rules that are used to derive the atoms for *at-cell/3* are instances of the rules:

$$\begin{aligned} at-cell(S_2, C, T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\ & take(Q_1, S_1, Q_2, S_2, D, T), \\ & reads(C, T), next(T, T') \end{aligned} \quad (6.5)$$

$$\begin{aligned} at-cell(S, C, T') \leftarrow & at-cell(S, C, T), \text{not } reads(C, T), \\ & cell(C), next(T, T'), symbol(S) \end{aligned} \quad (6.6)$$

Since there is only one atom *take/6* and one atom *reads/2* true for the time step k in C , there is only one ground instance of (6.5) where $T = k$, $T' = k + 1$, and the body is satisfied by $C(k)$. We use that instance to derive the atom *at-cell*($s', p, k + 1$) into $C(k + 1)$.

For all other cells it holds that *not reads*(p, k) is true, so (6.6) derives the atom *at-cell*($s, p, k + 1$) whenever *at-cell*(s, p, k) $\in C(k)$. By induction hypothesis there is only one such an atom for each cell, so $C(k + 1)$ will contain exactly one atom *at-cell/3* for every tape cell and the tape contents is the same as in the previous step except for the cell that was under the tape head that now contains the symbol that the transition wrote to it.

The only rule for *in-state/2* is:

$$\begin{aligned} in-state(Q_2, T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\ & take(Q_1, S_1, Q_2, S_2, D, T), next(T, T'), cell(C) . \end{aligned}$$

As there is only one atom for *take/6* with $T = k$ true in $C(k)$, this rule derives exactly one atom *in-state*($q', k + 1$) to $C(k + 1)$ where the new state corresponds to the transition that M took at c_k .

Finally, the rule for *reads/2* is:

$$\begin{aligned} reads(C', T') \leftarrow & \text{transition}(Q_1, S_1, Q_2, S_2, D), \\ & take(Q_1, S_1, Q_2, S_2, D, T), reads(C, T), \\ & next-cell(C, D, C'), time(T) . \end{aligned}$$

Again, there is only one instance whose body is satisfied so we get only one atom *reads*($p', k + 1$) $\in C(k + 1)$. The value of p' is defined by the predicate *next-cell/3* whose extension contains atoms of three forms:

$$\begin{aligned} & next-cell(p, s, p), \\ & next-cell(p, l, p - 1), \text{ and} \\ & next-cell(p - 1, r, p) . \end{aligned}$$

Thus, if the direction component of the transition is s , $p' = p$, if it is l , $p' = p - 1$, and if r , $p' = p + 1$. Thus, the tape head ends up in the cell that corresponds to the location at c_{k+1} . Here it is important to note that since M is time-bounded by $f(|x|)$, it cannot visit any cell that is further from the beginning than $f(|x|)$ so all possible cells are included in *cell/1* instances. Thus, we see that $C(k + 1)$ corresponds to c_{k+1} .

Finally, consider the rule for *accept*:

$$accept \leftarrow in-state(y, T), time(T) .$$

We see that we include *accept* in C exactly when M enters the state y . We showed that *in-state*/2 corresponds to the states that M visits during the computation, so we derive *accept* exactly when M accepts x . \square

Nondeterministic Simulation

Next, we argue that also the non-deterministic Turing machine simulation is correct.

Proposition 6.4.3 *Let $M = \langle K, \Sigma, \Delta, s \rangle$ be a nondeterministic Turing machine and $x \in \Sigma^*$. Then, $x \in L(M)$ if and only if $NTM(M, f(|x|))$ has a stable model C such that *accept* $\in C$.*

Proof. By Proposition 6.4.2 we know that $TM(M, n)$ simulates a deterministic Turing machine. The program $NTM(M, n)$ is otherwise the same as $TM(M, n)$ except that instead of having only one applicable ground instance for *take*/6 at any given time step, we now have a choice of more than one transition. The rules for *take*/6 are:

$$\begin{aligned} \{take(Q_1, S_1, Q_2, S_2, D, T)\} &\leftarrow transition(Q_1, S_1, Q_2, S_2, D), \\ &\quad in-state(Q_1, T), reads(C, T), \\ &\quad at-cell(S_1, C, T), time(T), cell(C) \\ &\leftarrow 2 \{ \{Q_1, S_1, S_2, S_2, D\}.take(Q_1, S_1, Q_2, S_2, D, T) : \\ &\quad transition(Q_1, S_1, Q_2, S_2, D), \\ &\quad time(T) \\ &\leftarrow not\ 1 \{ \{Q_1, S_1, S_2, S_2, D\}.take(Q_1, S_1, Q_2, S_2, D, T) : \\ &\quad transition(Q_1, S_1, Q_2, S_2, D), \\ &\quad time(T) \} \} \end{aligned}$$

Consider a stable model C of $NTM(M, n)$. Let the sets $C(k)$ be defined as in the proof of Proposition 6.4.2, and suppose that up to some i , the sets $C(n)$, $n \leq i$ correspond to one computation $\langle c_0, \dots, c_n \rangle$ of $M(x)$.⁴

Now there is again exactly one *at-cell*(σ, p, n) in $C(n)$ for every tape cell, one *reads*(p, n), and one *in-state*(q, n). Thus, the first rule for *take*/6 allows us to derive any atoms of the form *take*($q, \sigma, q', \sigma', d, n$) into C . The second rule forbids us from including two of them, and the third rule forces us to add at least one. Thus, we get exactly one atom *take*/6 into C . An argument similar to the one presented in the proof of Proposition 6.4.2 allows us to conclude that $C(n+1)$ corresponds to the configuration c_{n+1} .

If there are no atoms *take*/6 available, then the stable model candidate is discarded. This is the situation where a Turing machine has no applicable transitions. As the machine does not end in the accepting state, it is a rejecting computation so the behavior is correct. \square

6.5 PRELIMINARIES FOR COMPLEXITY PROOFS

In this section we build constructs that we will use in the complexity proofs. We divide our proofs into two parts, *inclusion* and *hardness*.

⁴The basic case of c_0 is identical with the deterministic one.

When proving inclusion, we show that the problem belongs to some complexity class. We establish it by showing that the size of the relevant instantiation of a program does not grow faster than some upper bound and that we can compute the domain model D_ω in a time that is polynomial with respect to the size of the instantiation.⁵ We use the Turing machine translation that was introduced in the previous section for the hardness proofs.

In this section we first define formally what the size of a program means and then we present algorithms for INSTANTIATION and MODEL. The algorithm for INSTANTIATION works in a time that is polynomial with respect to the size of the Herbrand instantiation, and that for MODEL is the nondeterministic variant of it.

6.5.1 Size of a Program

Before presenting the results we have to define what the size of a program means. There are several possibilities that we could use. For example, we could count the number of rules or the number of basic literals in the programs. There are situations where both above choices are suitable for practical purposes. However, with ω -restricted programs they do hide one source of complexity since also the size of the terms may increase when symbolic functions are used.

Definition 6.5.1 *The size of a term t is defined inductively as follows:*

1. $\text{size}(c) = 1$ for all 0-ary constants c ; and
2. $\text{size}(f(t_1, \dots, t_n)) = 1 + \sum_{i=1}^n \text{size}(t_i)$ for all n -ary terms $f(t_1, \dots, t_n)$, $n > 0$.

For an atom $A = p(t_1, \dots, t_n)$ or a negative literal $\text{not } A$:

$$\text{size}(A) = \text{size}(\text{not } A) = 1 + \sum_{i=1}^n \text{size}(t_i) .$$

The size of a conditional literal $\mathcal{L} = X.L : A$ is:

$$\text{size}(\mathcal{L}) = \text{size}(L) + \text{size}(A) .$$

For a cardinality atom $\mathcal{C} = \text{Card}(b, S)$:

$$\text{size}(\mathcal{C}) = \text{size}(\text{not } \mathcal{C}) = 1 + \sum_{\mathcal{L} \in S} \text{size}(\mathcal{L}) .$$

The size of a rule R is:

$$\text{size}(R) = \sum_{H \in \text{head}(R)} \text{size}(H) + \sum_{\mathcal{C} \in \text{body}(R)} \text{size}(\mathcal{C}) .$$

The size of a program P is:

$$\text{size}(P) = \sum_{R \in P} \text{size}(R) .$$

⁵We use a quadratic algorithm in this section but we can compute the domain model in linear time with respect to the instantiation using a Dowling-Gallier-type [51] algorithm [174].

Example 6.5.1 Consider the program P :

$$\begin{aligned} R_1 : & \quad a(1) \leftarrow \\ R_2 : & \quad b(f(X), X) \leftarrow a(X) \\ R_3 : & \quad c \leftarrow 1 \{ \{X, Y\}.d(X, Y) : b(X, Y) \} \quad . \end{aligned}$$

Here $\text{size}(R_1) = 2$, $\text{size}(R_2) = 6$, $\text{size}(R_3) = 8$, so $\text{size}(P) = 2+6+8 = 16$.

To make our proofs simpler we will overestimate the program sizes. We will assume that every literal and non-constant function symbol occurring in a program has the same number of arguments. In effect, we pad them by adding enough new arguments to make them to have the same arity as the term or predicate symbol that has the maximum number of them. Another assumption is that every rule has as many atoms occurring in the conditional literals in its body. Again, we can add irrelevant conditional literals to the rule bodies to make this so.

Definition 6.5.2 Let P be an ω -restricted program. Then, its approximated size is defined to be:

$$\text{size}_A(P) = m \cdot d \cdot \ell \cdot T \quad (6.7)$$

where:

- m is the number of rules in P ;
- d is the maximum arity of predicates and non-constant function symbols in P or 1, whichever is larger;
- ℓ is the number of atoms that occur in each rule; and
- T is the maximum size of a term that occurs in P .

Proposition 6.5.1 Let P be an ω -restricted program. Then,

$$\text{size}(P) \leq \text{size}_A(P) \quad .$$

Proof. Note that for every rule R , $\text{size}(R)$ is the sum of the sizes of atoms that occur anywhere in it. Let there be ℓ' such atoms and suppose further that the largest of them has d' arguments and that the largest term that occurs in it is of size T' . Thus, $\text{size}(R) \leq \ell' \cdot d' \cdot T' \leq \ell \cdot d \cdot T$ and $\text{size}(P) \leq \text{size}_A(P)$. \square If we can establish an upper bound for $\text{size}_A(P)$, then we know that it is also an upper bound for the size of P .

Sources of the Size Explosion

When we instantiate a program P , the size explosion of $\text{inst}_{\mathbf{H}_r}(P)$ comes mainly from two sources:

1. Creating new terms using function symbols with rules of the form:

$$p(f(X, Y)) \leftarrow d(X), d(Y)$$

If the extension of $d/1$ has c different ground terms, then instantiating this rule generates c^2 new ground terms that may then be used when instantiating rules that depend on $p/1$.

2. Constructing the Cartesian product of existing ground terms using rules of the form:

$$p(X_1, \dots, X_n) \leftarrow d(X_1), \dots, d(X_n) .$$

Instantiating a rule of this form results in t^n ground instances where t is the size of the extension of $d/1$.

There is one factor more that increases the size of the instantiated program but whose contribution is small compared with the two cases above:

3. When a function symbol f is applied to the terms t_1, \dots, t_n , then the size of the new term is the sum of the sizes of the argument terms.

6.5.2 Algorithms for INSTANTIATION and MODEL

We can solve INSTANTIATION with the algorithm presented in Figure 6.2. The algorithm is based on the function *naive-instantiate-relevant* from Section 5.7.

Proposition 6.5.2 *The algorithm $\text{instantiation}(P, A)$ returns **true** if and only if $D_\omega \models A$.*

Proof. We have already show in the proof of Theorem 5.0.2 that the set D that *instantiation* computes is the domain model of P . Since A is a ground atom, D satisfies A if and only if $A \in D$. \square

We will use this naive algorithm to prove inclusion results. The key point to note is that its time complexity is polynomial with respect to the size of the relevant instantiation of the program. Note that this does not give us a polynomial algorithm for INSTANTIATION since the instantiation may be exponential in size with respect to the size of the original program.

Proposition 6.5.3 *The time complexity of $\text{instantiation}(P, A)$ is polynomial with respect to the size of $\text{inst}_{\mathbf{Hr}}(P)$.*

Proof. By Theorem 4.6.1 we can compute a strict ω -stratification for P and check if it is ω -restricted in polynomial time with respect to the size of P , so the first part of *instantiation* is polynomial also with respect to $\text{inst}_{\mathbf{Hr}}(P)$ since $\text{size}(P) \leq \text{size}(\text{inst}_{\mathbf{Hr}}(P))$ for all P .

Next, we go through the finite strata of P in a while loop where we examine every strata once. The number of strata is linear with respect to $\text{Preds}(P)$ so this loop is executed a linear number of times. Inside the loop we have two operations, computing the relevant instantiation and finding its least model.

We can use the one-step provability operator to compute the least model of a stratum program P_i . The first step is that we remove all negative literals from its relevant instantiation $\text{inst}_{\mathbf{Hr}}(P_i, D_i)$. We can do this in linear time with respect to $\text{size}(\text{inst}_{\mathbf{Hr}}(P_i, D_i))$ using the construct from the proof of Theorem 5.1.1. Each iteration of T_P takes a linear time

```

function instantiation(Program  $P$ , Ground Atom  $A$ )
   $D := \emptyset$ 
   $\mathcal{S} := \text{create-stratification}(P)$ 
  if not is-restricted( $P, \mathcal{S}$ ) then
    return false
  endif
   $i := 0$ 
  while  $P_i$  is not empty do
     $PG := \text{naive-instantiate-relevant}(P_i, D)$ 
     $D := D \cup \mathbf{MM}(PG)$ 
     $i := i + 1$ 
  endwhile
  if  $A \in D$  then
    return true
  else
    return false
  endif
endfunction

function naive-instantiate-relevant(Ruleset  $R$ , Atomset  $D$ )
   $S := F(D)$ 
  foreach  $r \in R$  do
     $S' := \{r' \in \text{inst}(r, D) \mid D \models \text{body}_D(r')\}$ 
     $S := S \cup S'$ 
  end foreach
  return  $S$ 
end function

```

Figure 6.2: A naive algorithm for solving INSTANTIATION

with respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P_i, D_i))$. In the worst case every atom from $\text{inst}_{\mathbf{H}_r}(P_i, D_i)$ belongs to the least model and we have to derive each of them with a separate T_P operation. This means that this step takes $O(\text{size}(\text{inst}_{\mathbf{H}_r}(P_i, D_i))^2)$ time.

Finally, we need to examine *naive-instantiate-relevant*. In that function we have a loop that examines all rules in it one at a time and creates its relevant instantiation. We can do it by creating the relation \mathbf{R}_R that corresponds to the body of the rule R . To do this, we need to compute the relation $\mathbf{R}_{R, \mathcal{D}}$ that corresponds to the domain literals of R . We can do it in a time that is linear to the size of $R_{\mathcal{D}}$ and that in turn is linear with respect to $\text{inst}_{\mathbf{H}_r}(R)$. Note that we do not have to consider the complete instantiation of R since $\mathbf{R}_{R, \mathcal{D}}$ contains all those instances that satisfy the domain literals.

Putting all these steps together we see that the time to solve *instantiation*(P, A) is polynomial with respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P_i, D_i))$. \square

We can modify the *instantiation* algorithm to get a nondeterministic algorithm for MODEL. We instantiate the program, guess a stable model, and then verify that the model is correct. This takes time that is polynomial with respect to the size of the instantiation.

Proposition 6.5.4 *The time complexity for MODEL(P) is nondeterministic polynomial with the respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P))$.*

Proof. We can create the complete relevant instantiation of P analogously to the algorithm *instantiation*. By Proposition 6.5.3 this can be done in a time that is polynomial with respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P))$. Next, we nondeterministically guess a model candidate M . We can do this in a time that is linear with respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P))$. We then compute the reduct $\text{inst}_{\mathbf{H}_r}(P)^M$ and find its least model. Both of these operations can be done in a time that is polynomial with respect to $\text{size}(\text{inst}_{\mathbf{H}_r}(P))$. \square

6.6 COMPLEXITY RESULTS

Now, we go through the four different classes of ω -restricted programs one by one.

Theorem 6.6.1 *INSTANTIATION of an ω -restricted program is \mathbf{P} -complete when the number d of variables occurring in each rule is fixed and there are no non-constant function symbols in it.*

Proof. We construct the proof in two parts:

- (a) *Inclusion.* Let P be a program with d distinct variables. First we note that all ground terms that may occur in the relevant instantiation $\text{inst}_{\mathbf{H}_r}(P_i)$ of some stratum program P_i have to occur somewhere in P since non-constant function symbols are not allowed. Thus, each rule in P may have at most n^d ground instances where n is the number of different constants that occur in P . Similarly, the relevant expansion of a conditional literal may have at

most n^d basic literals in it so the total size of the instantiation is polynomial in the size of the program P and by Proposition 6.5.3 it can be computed in polynomial time.

- (b) *Hardness.* The problem of deciding whether an atom A belongs to the least model of a ground stratified normal logic program P is **P**-complete [38]. Such programs are a special case of ω -restricted programs (Proposition 3.6.1). Thus, it is **P**-hard to decide whether $D_\omega \models A$.

□

Theorem 6.6.2 *MODEL for an ω -restricted program with a fixed number d of variables and no non-constant function symbols is **NP**-complete.*

Proof.

- (a) *Inclusion.* As we saw in the previous proof, the instantiation of a fixed-variable ω -restricted program can be computed in polynomial time and it has a polynomial number of rules. By Proposition 6.5.4 we can determine whether it has a stable model in nondeterministic polynomial time.
- (b) *Hardness.* MODEL for ground normal logic programs is **NP**-complete [131] and they are a special case of ω -restricted programs (Proposition 3.6.1).

□

Theorem 6.6.3 *INSTANTIATION of an unlimited-variable ω -restricted program is **EXP**-complete if no non-constant function symbols are allowed.*

Proof.

- (a) *Inclusion.* Consider an ω -restricted program P and let $\text{size}_A(P) = n$. Suppose that there are c different ground terms in P . We now prove by induction over $d = 1, 2, \dots$ where d is the maximum number of variables in rules and conditional literals that the maximum size of the instantiation of P is:

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq m\ell c^{2d} \leq dn^{2n+2}$$

where m is the number of rules and ℓ the number of literals in each rule and $\text{size}_A(P) = n = m\ell$. (Since no new ground terms are generated during the instantiation, the maximum term length T is constant and may be left out of consideration.) We will also show that dn^{2n+2} grows asymptotically slower than 2^{n^2} to show that the problem is in **EXP**.

We divide our analysis in two parts. First, we show that the number of ground instances of rules in $\text{inst}_{\mathbf{H}_r}(P)$ is at most mc^d , and then we show that each such a rule may have at most ℓc^d ground literals in it.

Consider a single rule R . In the basic case $d = 1$ we see that it may have at most c instances that are obtained by substituting the sole variable by each ground term at a time. So,

$$|\text{inst}_{\mathbf{Hr}}(R)| \leq c .$$

Next, suppose that for all programs that have at most k different variables X_1, \dots, X_k in them, $|\text{inst}_{\mathbf{Hr}}(R)| \leq c^k$ for each rule R .

Consider the case where $d = k + 1$. Here we can do the instantiation of a rule R in two steps:

1. first instantiate only the variable X_{k+1} and leave the other variables still in place; and
2. instantiate variables X_1, \dots, X_k in the rules that were produced in the previous step.

The first step directly corresponds to the basic case $d = 1$, so we see that we can get at most c partially instantiated rules. By induction hypothesis, each one of them may have at most c^k ground instances and:

$$|\text{inst}_{\mathbf{Hr}}(R)| \leq c^k \cdot c = c^{k+1}$$

so we have established that a single rule may have at most c^d instances. As P has m rules, the total number of instantiated rules in $\text{inst}_{\mathbf{Hr}}(P)$ is:

$$|\text{inst}_{\mathbf{Hr}}(P)| \leq mc^d .$$

We can make exactly the same induction to show that the expansion of a conditional literal occurring in the rule body may contain at most c^d atoms in it, so the body of an instantiated rule may contain at most ℓc^d literals, each having the size d . Combining these two figures we get:

$$\text{size}(\text{inst}_{\mathbf{Hr}}(P)) \leq \text{size}_A(P) = (mc^d) \cdot (d\ell c^d) = md\ell c^{2d} \leq dn^{2d+2} .$$

The last inequality holds since $m \leq n$, $\ell \leq n$, and also necessarily $c \leq n$.

Since also d is linear to the size of n , the size of the ground instantiation grows $O(n^{2n+3})$. When we compare the growth rates of $f(n) = n^{2n+3}$ and $g(n) = 2^{n^2}$, we see that:

$$\begin{aligned} \lg f(n) &= (2n + 3) \lg n \\ \lg g(n) &= n^2 \end{aligned}$$

and conclude that $g(n)$ grows asymptotically faster than $f(n)$ since \lg is a monotonic function and n^2 grows faster than $(2n + 3) \lg n$. Thus, $n^{2n+3} = O(2^{n^2})$ and the size of the ground instantiation of P is bounded from above by an exponential function. Thus, INSTANTIATION is in **EXP**.

- (b) *Hardness.* First we note that a deterministic **EXP**-time Turing machine M uses at most 2^{n^k} time steps for some fixed k when the length of the input is n . We have to show that we can generate an exponential number of atoms representing time steps and tape cells using a program whose size is polynomial with respect to the size of M . To do this, we need to implement an n^k -bit binary counter that runs from 0 to $2^{n^k} - 1$. This can be done by encoding the numbers as vectors of binary variables:

$$number(X_1, \dots, X_{n^k}) \leftarrow bit(X_1), \dots, bit(X_{n^k}) . \quad (6.8)$$

The predicate $bit/1$ is auxiliary with the extension $\{0, 1\}$. The successor relation can be encoded with the rule:

$$\begin{aligned} next(X_1, \dots, X_{n^k}, Y_1, \dots, Y_{n^k}) \leftarrow & add(X_1, 1, Y_1, C_1), \\ & add(X_2, C_1, Y_2, C_2), \\ & \vdots \\ & add(X_{n^k}, C_{n^k-1}, Y_{n^k}, C_{n^k}) \end{aligned} \quad (6.9)$$

where $add/4$ encodes one-bit addition and is defined using the following four facts:

$$\begin{aligned} add(0, 1, 1, 0) \leftarrow & \quad add(0, 0, 0, 0) \leftarrow \\ add(1, 0, 1, 0) \leftarrow & \quad add(1, 1, 0, 1) \leftarrow \end{aligned} . \quad (6.10)$$

Now the time steps and tape positions can be defined in terms of numbers:

$$\begin{aligned} time(X_1, \dots, X_{n^k}) \leftarrow & number(X_1, \dots, X_{n^k}) \\ cell(X_1, \dots, X_{n^k}) \leftarrow & number(X_1, \dots, X_{n^k}) . \end{aligned} \quad (6.11)$$

Finally, we replace all references to $time/1$ and $cell/1$ by $time/n^k$ and $cell/n^k$ and add all necessary domain predicates to the rule bodies.

□

Theorem 6.6.4 *MODEL of an unlimited-variable ω -restricted program is **NEXP**-complete if no non-constant function symbols are allowed.*

Proof. We can prove inclusion as in Theorem 6.6.2. For hardness, we can use the nondeterministic Turing machine program and construct the instance facts as in the proof of Theorem 6.6.3. □

When we consider **INSTANTIATION** with unlimited function symbols, the binary counter construction in the hardness direction is quite convoluted. Thus, we prepare the way for it by proving a simpler result, namely, that **INSTANTIATION** is **EXP**-hard for such programs.

Lemma 6.6.1 ***INSTANTIATION** of a fixed-variable ω -restricted program P that uses non-constant function symbols is **EXP**-hard, if the number of variables in each rule is $d \geq 8$ and there are at least two different non-constant function symbols available.*

Proof. We need to construct a counter from 0 up to $2^{n^k} - 1$. We do this by encoding an m -bit binary number x as a function $b_1(b_2(\dots b_m(0)\dots))$ where b_i is f if the i th bit of x is 0 and t if it is 1. The m -bit binary numbers can be generated recursively from $(m - 1)$ -bit numbers by the following two rules:

$$\begin{aligned} \text{number}_m(t(X)) &\leftarrow \text{number}_{m-1}(X) \\ \text{number}_m(f(X)) &\leftarrow \text{number}_{m-1}(X) \end{aligned} \quad (6.12)$$

Here we need $m + 1$ different *number* predicates since otherwise the rules would not be ω -restricted. As the basic case of the recursion, we define one 0-bit number as:

$$\text{number}_0(0) \leftarrow . \quad (6.13)$$

The successor relation can also be defined recursively:

$$\begin{aligned} \text{next}_{i+1}(t(X), t(Y)) &\leftarrow \text{next}_i(X, Y) \\ \text{next}_{i+1}(f(X), f(Y)) &\leftarrow \text{next}_i(X, Y) \\ \text{next}_{i+1}(f(X), t(Y)) &\leftarrow \text{last}_i(X), \text{first}_i(Y) \end{aligned} \quad (6.14)$$

where $\text{last}_i/1$ and $\text{first}_i/1$ are defined as:

$$\begin{aligned} \text{last}_i(t^i(0)) &\leftarrow . \\ \text{first}_i(f^i(0)) &\leftarrow . \end{aligned} \quad (6.15)$$

The translation uses $7n^k + 3$ rules to create all n^k -bit numbers so we now have a polynomial reduction from **EXP**-time Turing machines to ω -restricted programs using only function symbols and the proof is complete. \square

Theorem 6.6.5 *INSTANTIATION of a fixed-variable ω -restricted program that uses non-constant function symbols is 2-EXP-complete, if the maximum number of variables occurring in a rule or a literal set is $d \geq 8$.*

Before we present the proof of the theorem we define four lemmas that help us manage the inclusion size of the proof. With these lemmas we break the size of the ground instantiation into its component pieces.

Lemma 6.6.2 *Let P be an ω -restricted program where the maximum arity of a function symbol is d and whose largest ground term has the size T . Then, the size of the largest term that occurs in $\text{inst}_{\mathbf{Hr}}(P)$ is bounded above by $2^s d^s T$ where s is the number of nonempty strata in P .*

Proof. We prove by induction over the number of strata that the largest ground term t_i that occurs in the relevant instantiation of the stratum program P_i is at most $2^i d^i T$.

As the basic case we note that all rules on the 0-stratum are ground, so $\text{size}(t_0) = T \leq 2^0 d^0 T$. Next, suppose that the claim holds for all strata up to k . As the maximum arity of a function symbol is d , the largest possible term on the $k + 1$ -stratum is:

$$t_{k+1} = f(\underbrace{t_k, \dots, t_k}_{d \text{ times}})$$

where f is a function symbol of maximal arity. We see that:

$$\begin{aligned} \text{size}(t_{k+1}) &= d \cdot \text{size}(t_k) + 1 \\ &\leq 2d \cdot \text{size}(t_k) . \end{aligned}$$

Here we use the induction hypothesis to note that:

$$2d \cdot \text{size}(t_k) = 2d \cdot 2^k d^k T = 2^{k+1} d^{k+1} T$$

and the induction is complete. \square

Lemma 6.6.3 *If P is an ω -restricted program such that at most c_i ground atoms are true in the union of answer sets $\bigcup_{j \in [0, i]} M_i$ of the first $i+1$ stratum programs P_j , then for each rule $R \in P_{i+1}$ it holds that $|\text{inst}_{\mathbf{Hr}}(R)| \leq c_i^d$ where d is the number of global variables in R .*

Proof. Each global variable that occurs in R has to occur also in a positive domain literal in the rule body that belongs to some stratum $j \leq i$ so there are at most c_j variable substitutions that satisfy it. As there are d variables, the rule has at most c_i^d ground instances with satisfiable bodies. \square

Lemma 6.6.4 *Let P be a cardinality constraint program with s nonempty strata with m rules each. Then, $\text{inst}_{\mathbf{Hr}}(P)$ has less than $2^{d^s} m^{d^{s+1}}$ ground rules if the number of variables $d > 1$.*

Proof. We prove a slightly stronger claim. Let c_i be the number of ground atoms that are true in the union of answer sets $\bigcup_{j \in [0, i]} M_i$ of the first $i+1$ stratum programs P_j . Then,

$$c_i \leq 2^{\sum_{j=0}^{i-1} d^j} \cdot m^{\sum_{j=0}^i d^j}$$

when $i > 0$ and $d > 1$. We prove this by induction over the number of strata in P . First, we note that

$$\begin{aligned} c_0 &\leq m \\ c_1 &\leq m \cdot m^d + m = m^{d+1} + m \\ &\leq 2m^{d+1} = 2^{\sum_{i=0}^0 d^i} \cdot m^{\sum_{i=0}^1 d^i} \end{aligned}$$

The value for c_0 comes from the fact that all rules on the 0-stratum are ground and for c_1 we note that there are m rules on 1-stratum and by Lemma 6.6.3 each of them may have at most c_0^d instances. Next, suppose that the claim holds for all strata up to some k .

Each of the m rules in the $k+1$ -stratum may have at most c_k^d instances, so:

$$c_{k+1} = m \cdot c_k^d + c_k \leq 2m \cdot c_k^d .$$

Next, we use the induction hypothesis to replace c_k by its upper bound and get:

$$\begin{aligned} c_{k+1} &\leq m(2^{\sum_{i=0}^{k-1} d^i} \cdot m^{\sum_{j=0}^k d^j})^d + 2^{\sum_{i=0}^{k-1} d^i} \cdot m^{\sum_{j=0}^k d^j} \\ &\leq 2m(2^{\sum_{i=0}^{k-1} d^i} \cdot m^{\sum_{j=0}^k d^j})^d . \end{aligned}$$

When we perform the outermost exponentiation we get:

$$\begin{aligned}
2m(2^{\sum_{i=0}^{k-1} d^i} \cdot m^{\sum_{j=0}^k d^j})^d &= 2m \cdot 2^{d \sum_{i=0}^{k-1} d^i} \cdot m^{d \sum_{j=0}^k d^j} \\
&= 2m \cdot 2^{\sum_{i=0}^{k-1} dd^i} \cdot m^{\sum_{j=0}^k dd^j} \\
&= 2m \cdot 2^{\sum_{i=0}^{k-1} d^{i+1}} \cdot m^{\sum_{j=0}^k d^{j+1}} \\
&= 2m \cdot 2^{\sum_{i=1}^k d^i} \cdot m^{\sum_{j=1}^{k+1} d^j} \\
&= 2^{1+\sum_{i=1}^k d^i} \cdot m^{1+\sum_{j=1}^{k+1} d^j} \\
&= 2^{\sum_{i=0}^k d^i} \cdot m^{\sum_{j=0}^{k+1} d^j}
\end{aligned}$$

and the induction is complete. We can simplify the final expression by noting that:

$$\sum_{i=0}^k d^i = \frac{1 - d^{k+1}}{1 - d} < d^{k+1}$$

when $d > 1$ so we get the upper bound $2^{d^s} \cdot m^{d^{s+1}}$ for the number of rules in $\text{inst}_{\mathbf{H}_r}(P)$. \square

Lemma 6.6.5 *The expansion of a conditional literal \mathcal{L} that occurs in a rule on the k -stratum of an ω -restricted program contains at most $2^{d^k} \cdot m^{d^{k+1}}$ basic literals where m is the number of rules per strata and d the number of local variables in \mathcal{L} .*

Proof. First we note that if \mathcal{L} has d local variables, then there are at most c_{k-1}^d basic literals in the expansion of \mathcal{L} where c_{k-1} is the number of ground atoms that can be derived using rules on the first k strata. By Lemma 6.6.4 we know that $c_k \leq 2^{d^k} \cdot m^{d^{k+1}}$. \square

Now we are ready to give our proof of Theorem 6.6.5.

Proof. [of Theorem 6.6.5]

- (a) *Inclusion.* Without a loss of generality we may assume that there are s strata with m rules each in a program P . Since the number of variables d is fixed in this case, we ignore it from the size definition and use $\text{size}_A(P) = n = sm\ell T$ where ℓ is the number of basic literals in each rule and T the size of the largest term occurring in P .

Let m' , ℓ' , and T' denote the number of rules, basic literals, and maximum term size of $\text{inst}_{\mathbf{H}_r}(P)$. By Lemmas 6.6.2–6.6.5 we know that:

$$\begin{aligned}
m' &\leq 2^{d^s} m^{d^{s+1}} \\
\ell' &\leq 2^{d^s} m^{d^{s+1}} \ell \\
T' &\leq 2^s d^s T .
\end{aligned}$$

Thus, the total size of the instantiation is:

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq 2^{d^s} m^{d^{s+1}} \cdot 2^{d^s} m^{d^{s+1}} \ell \cdot 2^s d^s T .$$

As each of s , m , ℓ , d , and T are less than n , we get that:

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq 2^{n^n} n^{n^{n+1}} \cdot 2^{n^n} n^{n^{n+1}} n \cdot 2^n n^n n .$$

When we rearrange the terms we see that:

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq 2^{2n^n+n} n^{2n^{n+1}+n+2} \leq n^{4n^{n+1}+n+2} \quad (6.16)$$

when $n \geq 2$. We now compare the growth rate of $f(n) = n^{4n^{n+1}+n+2} d^n$ to the growth rate of $g(n) = 2^{2^{n^2}}$:

$$\begin{aligned} \lg f(n) &= (4n^{n+1} + n + 2) \lg n \leq 5n^{n+1} \lg n \\ \lg(5n^{n+1} \lg n) &= \lg 5 + (n+1) \lg n + \lg \lg n \\ \lg g(n) &= 2^{n^2} \\ \lg \lg g(n) &= n^2 . \end{aligned}$$

As n^2 grows strictly faster than $n \lg n + \lg \lg n$, we know that

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) = O(2^{2^{n^2}})$$

so INSTANTIATION is in 2-**EXP**.

- (b) *Hardness.* We have to construct a binary counter running from 0 to $2^{2^{n^k}}$ for a given n^k . We follow the example of the proof of Lemma 6.6.1 and implement the numbers as terms. However, in this case we start with n rules:

$$\begin{aligned} \text{number}_0(0) &\leftarrow \\ &\vdots \\ \text{number}_0(n) &\leftarrow \end{aligned}$$

Then, we create the further levels of numbers using:

$$\text{number}_{i+1}(f(X, Y)) \leftarrow \text{number}_i(X), \text{number}_i(Y) . \quad (6.17)$$

The intuition is that number_1 is seen as an n^2 -base number, number_2 as $(n^2)^2 = n^4$ base, and so on. At the total we will have n^k different levels.

Let c_i be the size of the extension of $\text{number}_i/1$. Here we see that:

$$\begin{aligned} c_0 &= n \\ c_1 &= n^2 \\ c_2 &= (n^2)^2 = n^4 \\ &\vdots \\ c_i &= (c_{i-1})^2 = n^{2^i} \\ &\vdots \\ c_{n^k} &= n^{2^{n^k}} \end{aligned}$$

Thus, we can create $n^{2^{n^k}}$ different terms using $n^k + n$ rules. As $2^{2^{n^k}} \leq n^{2^{n^k}}$, we now see that it is enough to model all necessary

integers. Next, we have only to show that we can handle the arithmetic of the terms by defining the successor relation. We start by defining n ground facts for the basic case:

$$next_0(i, i + 1) \leftarrow$$

and continuing with a recursive definition:

$$\begin{aligned} next_{i+1}(f(X, Y), f(X, Z)) &\leftarrow number_i(X), next_i(Y, Z) \\ next_{i+1}(f(X, L), f(Y, F)) &\leftarrow first_i(F), last_i(L), next_i(X, Y) . \end{aligned}$$

Finally, we have:

$$\begin{aligned} first_0(0) &\leftarrow \\ last_0(n) &\leftarrow \\ first_{i+1}(f(X, X)) &\leftarrow first_i(X) \\ last_{i+1}(f(X, X)) &\leftarrow last_i(X) \end{aligned}$$

and the proof is complete. □

We now present a simple example program that illustrates the doubly-exponential growth of ground instantiation.

Example 6.6.1 *Consider the following program P with two variables:*

$$\begin{aligned} d_0(0) &\leftarrow \\ d_0(1) &\leftarrow \\ d_1(f(X, Y)) &\leftarrow d_0(X), d_0(Y) \\ d_1(t(X, Y)) &\leftarrow d_0(X), d_0(Y) \\ d_2(f(X, Y)) &\leftarrow d_1(X), d_1(Y) \\ d_2(t(X, Y)) &\leftarrow d_1(X), d_1(Y) \\ d_3(f(X, Y)) &\leftarrow d_2(X), d_2(Y) \\ d_3(t(X, Y)) &\leftarrow d_2(X), d_2(Y) . \end{aligned}$$

Each of the predicate symbols belong to a stratum by itself. The number of rules in the ground instantiations of the four strata are:

$$\begin{aligned} |inst_{\mathbf{H}_r}(P_0)| &= 2 = 2^{2^1-1} \\ |inst_{\mathbf{H}_r}(P_1)| &= 2 \cdot |inst_{\mathbf{H}_r}(P_0)|^2 = 2 \cdot 2^2 = 8 = 2^{2^2-1} \\ |inst_{\mathbf{H}_r}(P_2)| &= 2 \cdot |inst_{\mathbf{H}_r}(P_1)|^2 = 2 \cdot 8^2 = 128 = 2^{2^3-1} \\ |inst_{\mathbf{H}_r}(P_3)| &= 2 \cdot |inst_{\mathbf{H}_r}(P_2)|^2 = 2 \cdot 128^2 = 32768 = 2^{2^4-1} . \end{aligned}$$

If we add a fifth predicate $d_4/1$ and identical rules for it, then it will have $2 \cdot 32768^2 = 2,147,483,648 = 2^{31} = 2^{2^5-1}$ instances.

On the other hand, if we add a new ground fact $d_0(2) \leftarrow$ and a rule $d_{i+1}(h(X, Y)) \leftarrow d_i(X), d_i(Y)$ for each three other strata, then the sizes

of the instantiations will be:

$$\begin{aligned}
|\text{inst}_{\mathbf{H}_r}(P_0)| &= 3 = 3^{2^1-1} \\
|\text{inst}_{\mathbf{H}_r}(P_1)| &= 3 \cdot |\text{inst}_{\mathbf{H}_r}(P_0)|^2 = 3 \cdot 3^2 = 27 = 3^{2^2-1} \\
|\text{inst}_{\mathbf{H}_r}(P_2)| &= 3 \cdot |\text{inst}_{\mathbf{H}_r}(P_1)|^2 = 3 \cdot 27^2 = 2187 = 3^{2^3-1} \\
|\text{inst}_{\mathbf{H}_r}(P_3)| &= 3 \cdot |\text{inst}_{\mathbf{H}_r}(P_2)|^2 = 3 \cdot 2187^2 = 14,348,907 = 3^{2^4-1} .
\end{aligned}$$

Thus, we see that if there are n strata and n rules in each stratum, then there are n^{2^k-1} ground instances on the k th stratum, so there are in total:

$$\sum_{k=1}^n n^{2^k-1}$$

instances. In this sum the final term n^{2^n-1} dominates the value and we can say that it grows $O(n^{2^n})$.

Theorem 6.6.6 MODEL for a fixed-variable ω -restricted program that uses function symbols is 2-NEXP-complete, if $d \geq 8$.

Proof. As in Theorem 6.6.4. □

Theorem 6.6.7 INSTANTIATION of an ω -restricted program is 2-EXP-complete.

Proof.

- (a) *Inclusion.* By Theorem 6.6.5 $\text{size}_A(\text{inst}_{\mathbf{H}_r}(P))$ is bounded from above with:

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq n^{4d^{n+1}+1}$$

The same result holds also in this case, but here d is not fixed but is linear to n . Thus,

$$\text{size}_A(\text{inst}_{\mathbf{H}_r}(P)) \leq n^{4n^{n+1}+1}$$

and further:

$$\begin{aligned}
\lg n^{4n^{n+1}+1} &= (4n^{n+1} + 1) \lg n \leq (5n^{n+1}) \lg n \\
\lg(5n^{n+1} \lg n) &= \lg 5 + (n+1) \lg n + \lg \lg n .
\end{aligned}$$

As $n \lg n + \lg \lg n = O(n^2)$, we see again that $\text{size}(\text{inst}_{\mathbf{H}_r}(P)) = O(2^{2^{n^2}})$ and the proof is complete.

- (b) *Hardness.* Follows directly from Theorem 6.6.5. □

Theorem 6.6.8 MODEL for an ω -restricted program is 2-NEXP-complete.

Proof. As in Theorem 6.6.4. □

6.7 THE FUNCTION VERSION OF MODEL

In most cases we are not content to find out whether a CCP has a stable model or not. Instead, the atoms that are true in a stable model give us the answer to our problem. This means that we are usually solving the function version of MODEL.

PROBLEM 3: FUNCTIONAL MODEL. Given a cardinality constraint program P , either find a stable model $M \in \mathbf{SM}(P)$ or prove that one does not exist.

As MODEL is **NP**-complete for ground CCPs, FUNCTIONAL MODEL is **FNP**-complete⁶ for them. What this means in practice is that all currently known algorithms for it have an exponential worst-case behavior. However, experiments have shown that the typical instances for **NP**-complete problems are easy to solve and the computationally hard cases are relatively rare [29, 139]. In this section we examine how we can estimate the effort that we have to expend to compute answer sets.

Many different algorithms [174, 5, 119, 80, 2, 39] have been presented for computing answer sets of logic programs under different semantics. This is an important research area that cannot be treated adequately within the space of this work. Instead, we work on an abstract level and treat the algorithms as determinized Turing machines that solve **NP**-problems. The idea is to characterize the difficulty for finding a stable model in terms of the nondeterministic choices that a program has to make while constructing the stable model.

Definition 6.7.1 *A CCP-solver A is an algorithm that takes an ω -restricted cardinality constraint program P as its input and returns either*

1. *a stable model $M \in \mathbf{SM}(P)$ if P is satisfiable; or*
2. *false if P is unsatisfiable.*

6.7.1 Determinizing Turing Machines

It is a well-known fact that we can always transform a nondeterministic Turing machine into a deterministic one [199]. We go systematically through all possible computations of the machine, examining every combination of nondeterministic choices one at a time. The price of removing guesses from an algorithm is an exponential increase of computation time. If a nondeterministic Turing machine makes n binary choices when processing the input x , there are 2^n possible computations that the corresponding deterministic machine has to examine with the same input.

We can visualize the computations of a nondeterministic Turing machine M as a tree where the configurations of the machine form the nodes and a node c' is a successor of a node c if $c \vdash_M c'$. These trees are usually called *computation trees*. If a node has two or more successors we call it a *choice point*. A computation tree is *complete* if it contains all

⁶Problems that are **FNP**-complete are often called **NP** search problems.

configurations that M may reach with a given input, and it is *accepting* if it contains a branch that leads into an accepting state. In the context of function problems we use the terms *search tree* and *solution* [167] to describe computation trees and accepting branches, respectively.

6.7.2 Computing FUNCTIONAL MODEL

The simplest nondeterministic algorithm for FUNCTIONAL MODEL is to guess the truth value for every ground atom, and then check whether the candidate is a stable model or not. The algorithm *has-stable-model* from Figure 5.1 in Section 5.7 that examines every subset of atoms in turn is essentially the determinized version of this idea.

This algorithm is not suitable for practical use. The problem is that the search space is far too large. A program whose relevant instantiation has n ground atoms has 2^n possible truth assignments. Most non-trivial programs will have thousands of atoms, and there exist examples with hundreds of thousands or even millions of atoms. It would be completely unfeasible to try to solve such instances if we always had to examine every possible combination. Practical solvers work by combining making choices with truth value propagation.

A *partial solver*⁷ is an algorithm that takes as its input a program P and a partial truth assignment $I = \langle I^+, I^- \rangle$ where I^+ contains those atoms that are set true, I^- those that are false, and the rest of the atoms are undefined. It then uses some set of propagation rules to extend I to cover more atoms.

Definition 6.7.2 A partial CCP-solver A is an algorithm that takes as its input a CCP P and a partial truth assignment $\langle I^+, I^- \rangle$ where $I^+ \cap I^- = \emptyset$, and returns a pair $\langle M^+, M^- \rangle$ where

1. $I^+ \subseteq M^+$ and $I^- \subseteq M^-$; and
2. for all stable models $M \in \mathbf{SM}(P)$ it holds that if $I^+ \subseteq M$ and $I^- \cap M = \emptyset$, then $M^+ \subseteq M$ and $M^- \cap M = \emptyset$.

We can think of the computation of an ASP solver as a procedure where we first guess the truth value of an atom and then call a partial solver to compute the consequences of this choice. If the sets M^+ and M^- that it returns are not disjoint, we have a contradiction where some atom has to be at the same time both true and false, so we reject our choice and try the other value. If M^+ and M^- together cover all atoms of the program, we have found an answer set. If neither of these things happen, we continue by guessing the value of the next atom.

In this work we follow the convention of Williams et. al. [209] and consider partial solvers that use propagation rules that can be computed in polynomial time. Practical ASP solvers usually use only rules that can be computed in linear or quadratic time [174].

⁷Partial solvers are also called sub-solvers.

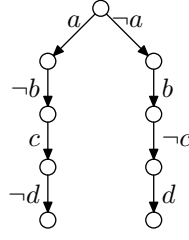


Figure 6.3: The search tree from Example 6.7.1

Example 6.7.1 Consider the program P :

$$\begin{aligned} a &\leftarrow \text{not } b \\ b &\leftarrow \text{not } a \\ c &\leftarrow a \\ d &\leftarrow b . \end{aligned}$$

Suppose that we have a partial solver that uses the following two propagation rules:

- (1) If the body of a basic rule is true, make its head true; and
- (2) If all rules for an atom have provably unsatisfied bodies, make it false.

This program has four atoms, so the size of the search space is $2^4 = 16$ truth assignments. However, we have effectively only one choice here.

Suppose that we make the guess that a is true. From this we can infer:

1. not b using (2);
2. c using (1); and
3. not d using (2).

Choosing one truth value was enough to determine the truth values of all other atoms. We can check that $\{a, c\} = M = \mathbf{MM}(P^M)$ so it is really a stable model.

The same thing happens if we start by guessing that a is false. Then, (1) allows us to infer b and then d , and (2) lets us conclude that c is false.

Figure 6.3 shows the search tree for this example. If we had made the initial choice over b , we would have got a search tree that is isomorphic to the given one.

If we make the initial choice over c or d , we cannot propagate any information and have to make additional choices.⁸ This shows that the order of choices has an impact on the search tree size.

⁸This is a limitation of our choice of propagation rules. If we include rules that allow backwards propagation from the head of the rule to the literals in the body [174], then choosing the value for c or d is also enough to completely determine the stable model.

6.7.3 Effective Search Space Sizes

Next, we examine how we can limit the number of nondeterministic choices that we need to make.

Williams et. al. [209] proposed the concept of *backdoor variables* for describing the complexity of propositional logic SAT problems.⁹ A set of propositional atoms is a backdoor for a SAT instance if setting their truth values allows us to solve the remaining subproblem in a polynomial time.

For example, the set $\{a\}$ is a backdoor for the program in Example 6.7.1 in this sense. After we chose a truth value for a , we could find the truth values for other atoms in a linear time. Different solvers use different propagation rules so they can have different sets of backdoors.

Definition 6.7.3 *Let P be a ground CCP and A a partial CCP-solver. A set $B \subseteq \text{Atoms}(P)$ is a backdoor for A in P if and only if for each subset $I \subseteq B$ it holds that $A(I, B \setminus I) = \langle M^+, M^- \rangle$ where M^+ is a stable model.*

This definition corresponds to the notion of *strong backdoor* of [209].¹⁰

Backdoors give us a better worst-case estimate on how difficult it is to find a stable model for a program P . If a program P has an m -atom backdoor for A and the solver is aware of the backdoor, then we can solve P in $O(2^m)$ time instead of $O(2^n)$ where n is the number of all atoms. We make the choices over the atoms in the backdoor and then call A to compute the truth values of the rest of the atoms in polynomial time.

Unfortunately, finding a backdoor is not easy. Deciding whether a SAT instance has a backdoor of size m is **NP**-hard [45] in the general case. However, there are several classes of backdoors that we can identify syntactically from a program.

Proposition 6.7.1 *Let P be a finite ground CCP and $S \subseteq P$ be the least set containing:*

1. *every atom A that occurs in the head of a choice rule in P ; and*
2. *every atom B that occurs in a negative literal in P .*

Then, S is a backdoor for some partial CCP-solver A .

Proof. We will now describe the partial CCP-solver A for which S is a backdoor. The partial solver takes as its arguments P and the partial truth assignment $\langle M, S \setminus M \rangle$. It works in two phases:

1. compute the reduct P^M ; and
2. compute the least model $M' = \mathbf{MM}(P^M)$.

⁹In SAT we want to find out if a propositional logic formula is satisfiable or not. For the formal definition see Section 9.2.1.

¹⁰A weak backdoor in SAT is a set of atoms that have at least one truth assignment that allows us to solve the resulting problem in polynomial time.

The set M' is a stable model of P if it holds that $M \subseteq M'$ and $M' \cap (S \setminus M) = \emptyset$. To see this note that M and M' agree on the atoms of S so the reduct $P^{M'} = P^M$. Thus, $M' = \mathbf{MM}(P^M) = \mathbf{MM}(P^{M'})$.

We can create the reduct P^M in linear time with respect to the size of P and find its least model in $O(n^2)$ time with the provability operator T_{P^M} . \square

We can get a slightly smaller backdoor by leaving out from S those negative atoms B that do not occur in a negative dependency cycle [174]. When we compute the reduct we do it only for the atoms that occur in S and leave all other negative literals in place. This leaves us with a stratified program that has unique least model that we can compute in quadratic time [144].

Corollary 6.7.1 *We can decide whether a finite ground CCP has a stable model in $O(2^m)$ where m is the number of atoms that occur either in the heads of choice rules or in negative literals that take part in negative cycles of the dependency graph of the program.*

This gives us a heuristics for writing programs: we can lower the worst case complexity for model generation by reducing the number of atoms in the set S .

The backdoor that we get this way is not necessarily the smallest backdoor that a program has. For example, for the program in Example 6.7.1 we get $S = \{a, b\}$ while each atom is a backdoor by itself.

6.7.4 More on Choice Points

The size of a backdoor gives us an upper bound for the size of the search tree of the program when the solver that we are using recognizes it. This tells us only how much effort we have to expend in the worst case. In the extreme case where all truth assignments of the backdoor lead to a stable model we can find one in a linear time.¹¹

The order in which we make the choices can have a large impact in computational efficiency. It is possible that with some orders we get an exponentially larger search tree than with other orders [34]. Also, even if we have identified a backdoor it may be useful to make choices over atoms that do not belong to it because sometimes choosing the truth value of a non-backdoor atom may in its turn fix the values of more than one atom in the backdoor [105].

There is a difference in finding all answer sets or just one of them. If we want to find only one answer set, we can stop after encountering the first solution while we have to explore the whole search tree if we want to get all answer sets. If we search for one answer set, we can get lucky and make a good guess early in the computation and enter a branch where we find a model quickly. Also, we might possibly use some propagation rules that preserve only satisfiability of the program and not all answer sets. With these kinds of rules we could conceivably find one answer set quickly at the cost of losing all other answer sets.

¹¹Supposing that the propagation rules can be executed in a linear time.

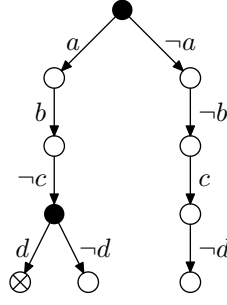


Figure 6.4: A search Tree for Example 6.7.2

Example 6.7.2 Consider the program:

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

Here we have a backdoor $\{a, d\}$, the atoms that occur in the heads of the choice rules. Even though b and c occur negatively, we do not have to include them in the backdoor since the negations are stratified.

We see a complete search tree in Figure 6.4. The black inner nodes correspond to choice points and white to points where the truth value is determined by earlier choices using the two propagation rules from Example 6.7.1.

When there are two atoms in the backdoor, the maximum size of the search tree is $2^2 = 4$ branches. In this case we have only three branches since we have a choice on the value of d only if b is true.

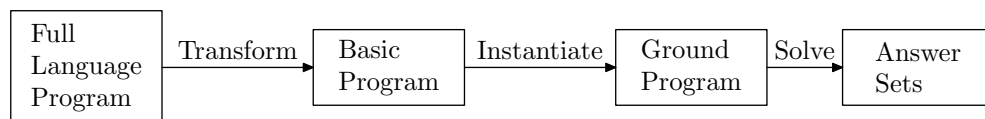


Figure 7.1: The process of computing answer sets

7 THE FULL LANGUAGE

When we write actual answer set programs, we want to have a language that has enough expressive power to allow us to use simple and natural encodings. However, when we are writing a solver engine that actually computes the answer sets, we want the language to be relatively simple so that we can create a more efficient implementation.

The way how we address this dilemma in this work is to divide the language into two parts:

1. A *basic language* that contains the basic primitives; and
2. A *full language* that provides a richer feature set for to the programmer.

The programmer uses the features of the full language to write encodings for problems. The ASP system then translates the program into a basic language program that is then given as input to the solver. This process is illustrated in Figure 7.1 and we see more on this approach in Chapter 8.

The Basic Language

We use the ω -restricted cardinality constraint programs that we have defined in the previous chapters as our basic language and every full language program will be translated into some set of ω -restricted rules.

The basic language takes a role that is similar to the role of internal representation in standard compiler technology [1]. There we have a front-end that reads in an input program and creates its internal representation, and then a back-end creates the object code from it.

The Full Language

The full language adds several features to the basic CCPs that we have found to be useful in expressing a wide variety of problems. We do not define a separate semantics for the full language but instead we give it in the terms of the translation to basic programs. A set of ground atoms is a stable model of a full program if and only if it is a stable model of the translated program.

7.1 LANGUAGE DESIGN

We want to construct the languages so that there is a close connection between the full and basic language. The basic language should have

primitives that allow us to implement the features of the full language in a concise and intuitively clear manner.

When considering what language constructs to include as primitives in the basic language and what we leave as extensions we had four principal criteria in mind:

1. we should be able to translate full programs to basic programs on the level of programs with variables;
2. the translations should be modular;
3. the translations should be concise; and
4. the translations should not increase the size of the Herbrand base of the instantiated program.

Our transformations satisfy these criteria quite well, but it turned out that there were two constructs where we had to make a compromise between our goals of keeping the basic language simple and the transformations elegant. The two problematic cases were upper bounds for negative cardinality literals (Section 7.3.2) and integral ranges (Section 7.3.6).

7.1.1 On Importance of Variables and Modularity

With ASP we want to use *uniform* encodings [171, 60, 130] as much as possible. The basic intuition of such an encoding is that we have one program with variables that we can use to solve all instances of a problem. Each instance is defined by a set of facts that combined with the rules of the encoding give its answer.¹ This idea is shown in Figure 7.2.

When we work with such encodings, we usually do not know what specific problem instances we want to solve so we have only the rules with variables available to us. We want to be able to intuitively understand what the rules of a program mean even when we do not have any ground instances to examine.

We also want to be able to compose the problem encodings from smaller pieces in a modular fashion [151] and to replace a program fragment with an equivalent one without having to worry about the effects of ground instances and extensions of domain literals.

This idea is closely connected with *modularity of transformations* [100]. A transformation τ is modular if $\tau(P_1 \cup P_2) = \tau(P_1) \cup \tau(P_2)$ for every program P_1 and P_2 . If our translation is modular, then we can compute it in parts. We first translate the uniform encoding, and then when we get the instance facts we translate them separately.² If a transformation is not modular, then we have to add the facts to the encoding before doing the translation. The differences between these two approaches are illustrated in Figures 7.3 and 7.4.

The transformations that we use are modular on the level of individual rules. That is, we can translate the program one rule at a time. If we

¹The nature of these encodings is examined in detail in Chapter 9.

²Though, in practice we do not have to translate facts since they are already given as basic programs.

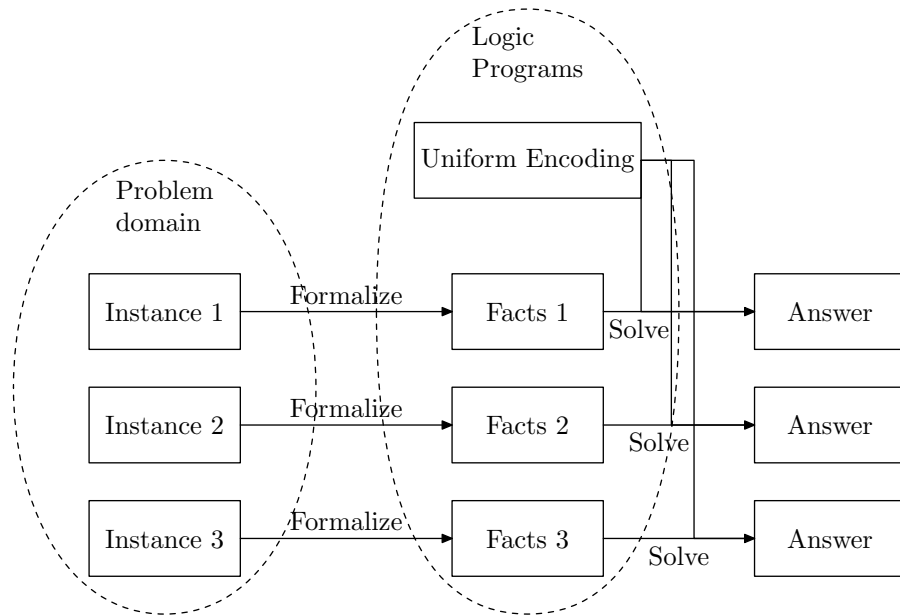


Figure 7.2: The principle of uniform encodings

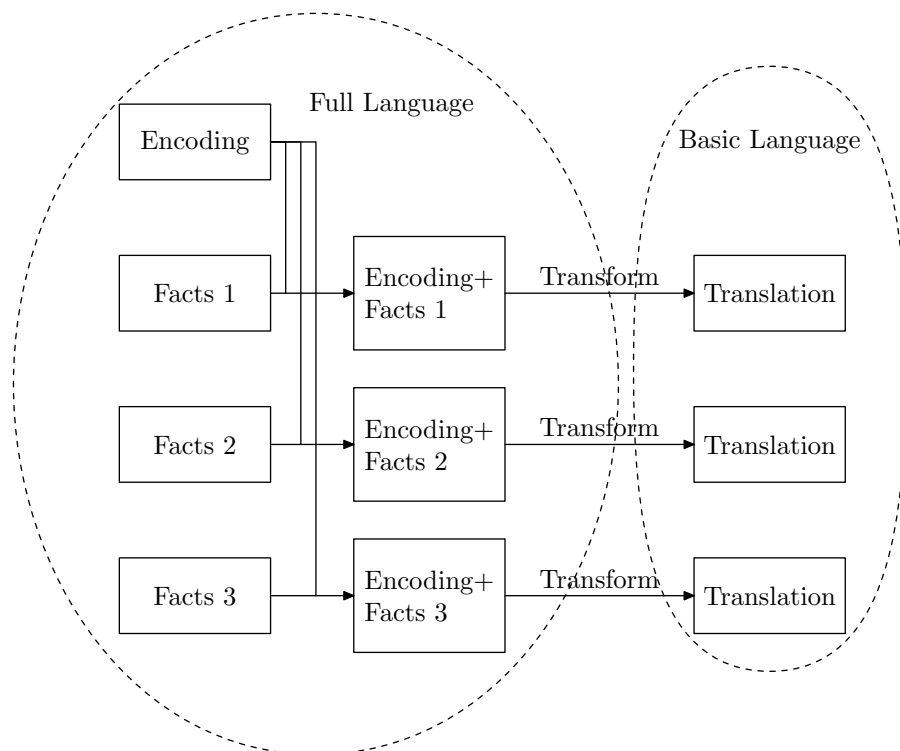


Figure 7.3: A non-modular transformation

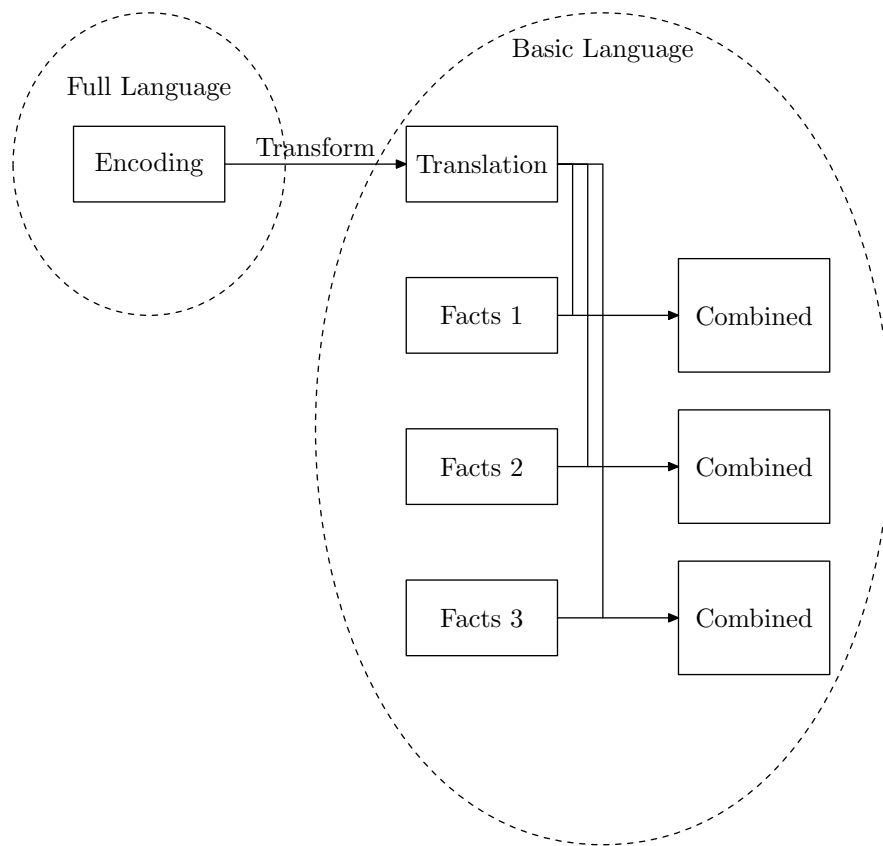


Figure 7.4: A modular transformation

then want to replace one rule with another, we need to transform only that rule to get the basic program.

7.1.2 On the Size of the Translation

Our size-related criteria come from practical considerations. A translation should not significantly increase the size of the program or its Herbrand base.

The consideration for the Herbrand base comes from the fact that each new atom that we add to it doubles the number of interpretations of the program and in the worst case also the time that it takes us to find a stable model also doubles. In practice the increase is usually not so great but generally the fewer relevant instances the non-domain predicates have, the faster it is to compute models.

With domain predicates the increase is not as drastic as long as we do not use them to create new terms. This is because the domain model is simple to compute compared with computing the models of the ω -program and adding a new domain predicate will not usually increase the size of the Herbrand base of the non-domain part of the program. This means that in our transformations we are willing to introduce new domain predicates since the benefits of a clearer transformation and simpler representation are in practice greater than the possible losses in computational efficiency.

Most of our transformations add a constant number of new literals or rules to the program. There are two exceptions for this. We need a linear number of new rules when we remove full cardinality atoms from the heads of rules, and when we transform a negative cardinality literal that has an upper bound, we have to make a choice of whether we want to add a linear number of new atoms into the Herbrand base of the program or potentially an exponential number of new rules. We examine this further in Section 7.3.2.

7.1.3 On Domain Predicates

In this work we demand that all variables occur in domain literals in the rule bodies. This is more limited than the standard range-restriction [142] that demands only that each variable should occur in some positive domain literal. This approach has the advantage that it makes it possible to compute a reasonably small instantiation in a reasonably fast time. This will be examined further in Section 8.4.3

How do we define and use domain predicates in practice? With uniform encodings we have one program that encodes the constraints of the problem and the individual instances are given as sets of facts. In the vast majority of cases the predicates that are defined in the instance files are domain predicates. We often want to use them to define additional domain predicates.

In a way, we can think that domain predicates assign types to variables. Logic programs have traditionally had little or no type checking and it is the responsibility of the programmer to ensure that the predicates are

Union $D_1 \cup D_2$:	$union(X) \leftarrow d_1(X)$ $union(X) \leftarrow d_2(X)$
Intersection $D_1 \cap D_2$:	$intersect(X) \leftarrow d_1(X), d_2(X)$
Difference $D_1 \setminus D_2$:	$diff(X) \leftarrow d_1(X), \text{not } d_2(X)$

Table 7.1: Set Operations on Domains

Composition $R_1 \circ R_2$:	$comp(X, Z) \leftarrow r_1(X, Y), r_2(Y, Z)$
Projection:	$proj(X, Y) \leftarrow r(X, Y, Z)$
Transitive closure R^* :	$tc(X, Y) \leftarrow r(X, Y)$ $tc(X, Z) \leftarrow r(X, Y), tc(Y, Z), d(Z)$
Symmetric closure $R \cup R^-$:	$sc(X, Y) \leftarrow r(X, Y)$ $sc(X, Y) \leftarrow r(Y, X)$

Table 7.2: Operations on Relations

used correctly. Domain predicates that are used as type predicates give us a rudimentary type system for ASP languages. The *lpars* instantiator has a support for domain declarations. For example, the declaration:

$$\#domain\ d(X)$$

defines a domain for the variable X . This is implemented by adding the literal $d(X)$ to every rule that contains the variable X .

A programmer can construct complex domains from simple ones with rules. Table 7.1 shows how to implement the basic set operations on domains and Table 7.2 do the same for relations.

Example 7.1.1 *Consider the problem of computing the transitive closure of a relation $R \subseteq D \times D$. If we are given both the relation R and the set D as facts of predicates $r/2$ and $d/1$, then we can compute the transitive closure with the rules:*

$$\begin{aligned} tc(X, Y) &\leftarrow r(X, Y) \\ tc(X, Z) &\leftarrow r(X, Y), tc(Y, Z), d(Z) \ . \end{aligned}$$

Here $r(X, Y)$ gives types for X and Y in both rules and $d(Z)$ does the same for Z . The predicate $tc/2$ is also a domain predicate and can be used in other rules.

If $d/1$ is not given as facts, we can define it directly from $r/2$ with the rules:

$$\begin{aligned} d(X) &\leftarrow r(X, Y) \\ d(Y) &\leftarrow r(X, Y) \ . \end{aligned}$$

7.1.4 On Literal Types

We have two types of negative literals in our basic language which raises the question of whether both of them are necessary or if one of them can be expressed in the terms of the other. On the surface it seems that they are redundant since for all ground cardinality literals it holds that:

$$M \models \text{Card}(b, S) \text{ iff } M \models \text{not Card}(b^M, \{\overline{\mathcal{L}} \mid \mathcal{L} \in S\})$$

where $b^M = |S| - b + 1$. For example, the cardinality literal

$$\text{not Card}(1, \{b, \text{not } c, \text{not } d\})$$

is satisfied by the same sets of atoms as:

$$\text{Card}(3, \{\text{not } b, c, d\}) .$$

However, we can do this translation only for ground programs. If we have a cardinality literal $\mathcal{C} = \text{not Card}(b, \{X.a(X) : d(X)\})$, we do not know how many atoms belong to the set before we have instantiated $d(X)$. Thus, we cannot replace \mathcal{C} by an equivalent positive literal when we work on the level of uniform encodings.

Conditional Literals

A conditional literal $X.p(X) : q(X)$ allows us to define a set of literals in a compact manner. These literals were originally introduced in the *lp* version 0.99.30 [197]. In Section 7.3.4 we extend the syntax to allow a conjunction of literals in the condition.

The *dlv* system [111] has *symbolic sets* that can be used the same purpose. A symbolic set has the form $\{X : S\}$ where X is a list of local variables and S is a conjunction of literals. Intuitively, a symbolic set works like a conditional literal with the difference that it is replaced by a set of tuples instead of a set of basic literals. Thus, $\{X : p(X) \wedge q(X)\}$ behaves in the same way as $X.p(X) : q(X)$ does. However, the semantics is not the same since symbolic sets can represent also multisets instead of sets. When they are used in this capability their semantics is difficult to capture with conditional literals.

Parametric connectives [155] are another similar construct. A parametric connective is either a disjunction $\bigvee \{L : C\}$ or a conjunction $\bigwedge \{L : C\}$ where L is a basic literal and C is a conjunction of basic literals.

The intuitive meaning of $\bigvee \{p(X) : q(X)\}$ is that it is instantiated into a disjunction $p(t_1) \vee \dots \vee p(t_n)$ where $q(t_i)$ is true for all t_i . Thus, it behaves in a similar way to a cardinality atom $\text{Card}(1, \{X.p(X) : q(X)\})$. However, in [155] a parametric disjunction may occur only in a rule head and not in the body.

We do not allow cardinality atoms in the head of a rule in our basic language but in this section we will extend our syntax to include them in the full language. With this we can simulate the syntax of parametric disjunctions. However, the semantics is different since the parametric disjunctions use the semantics of disjunctive Datalog [60].

A parametric conjunction $\bigwedge \{p(X) : q(X)\}$ is instantiated in the same way as the disjunction except that the resulting literals are added to the rule body. This acts as a universal quantification $\forall X. q(X) \rightarrow p(x)$. For example, if we have:

$$h \leftarrow \bigwedge \{p(X) : q(X)\}$$

where the extension of $q/1$ is $\{q(0), q(1)\}$, then the rule is instantiated as:

$$h \leftarrow p(0), p(1) .$$

If the extension of $q/1$ is empty, then the rule instantiates as a fact for h .

We allow conditional literals to be used in a similar way in the full language. A conditional literal $X.p(X) : d(X)$ that occurs directly in a rule body is seen to represent a cardinality atom that is true when $p(X)$ is true for every X for which $d(X)$ is true. We examine this construct in more detail in Section 7.3.5.

7.1.5 On Rule Types

We have three types of rules in the language, basic and choice rules and constraints:

$$\begin{aligned} a &\leftarrow body \\ \{a\} &\leftarrow body \\ &\leftarrow body \end{aligned}$$

We chose to include basic and choice rules as primitives and then implement constraints in terms of basic rules. This is not the only possibility since it is possible to express a basic rule with a choice rule and a constraint:

$$\begin{aligned} \{a\} &\leftarrow body \\ &\leftarrow body, \text{not } a . \end{aligned}$$

With this approach we could define a stable model so that a set is a stable model if it is the least model of the reduct and it additionally satisfies all constraints. Our choice of having basic rules instead of constraints is largely pragmatic as in general it is easier to handle single rules than pairs of rules.

If we are willing to add new atoms to the program we do not need choice rules and can do with basic rules only. If we have a rule:

$$\{a, b\} \leftarrow body ,$$

then the corresponding basic rule construct is:

$$\begin{aligned} a &\leftarrow \text{not } a', body \\ a' &\leftarrow \text{not } a \\ b &\leftarrow \text{not } b', body \\ b' &\leftarrow \text{not } b . \end{aligned}$$

The problem here is that we will essentially double the size of the ground instantiation of a program. Choice rules are convenient from the knowledge-representation point of view because in many if not most problems their semantics corresponds to the intuitive concept of choices. We have found that students who are introduced to ASP find it easier to use choice rules than the traditional even-loop construction in creating the programs.

7.1.6 On Aggregates

Cardinality literals are one form of more general *aggregate literals* [107, 42, 43, 129, 69, 126, 183, 154, 182, 65, 67]. An aggregate literal combines the truth values of a set of literals into one aggregate value. In most cases the aggregates are defined in terms of weighted literals. We will see one full example of such an aggregate later in Section 7.4.1 where we will introduce weight atoms. In this discussion we will use the set of literals $S = \{a_1, a_3, a_5, a_7\}$ where each atom a_i has the weight i .

Possible aggregate types include maximum, average, and parity, among others. A maximum atom $\max(b, S)$ is satisfied if the maximum weight among the true literals in S exceeds the bound b . For example, the set $\{a_3, a_5\}$ satisfies the atom $\max(4, S)$, but $\{a_1, a_3\}$ does not. A parity aggregate atom $\text{even}(S)$ is satisfied if an even number of literals in S are true. For example, $\{a_1, a_5\}$ satisfies it but $\{a_1\}$ does not. An average aggregate behaves similarly to $\max/2$ except that it uses the average weight instead of the maximum.

It is also possible to use other relations in place of greater-than. For example, we might say that a cardinality atom is true if exactly three literals in it are true. In fact, some definitions [129, 126] allow the use of arbitrary sets in the aggregates.

What is the reason that cardinality constraint programs are based on specifically on cardinality atoms and not other aggregates? Part of the reason is historic since they are the first form of aggregate literals that were implemented for an answer set programming system [179]. Another reason is that they are well-behaved. In Chapter 3 we could use the traditional definition [204] of the one-step provability operator T_P directly for programs that contained cardinality atoms. This property is not satisfied by all possible aggregates. To see why this happens consider the following example that uses parity aggregates:

Example 7.1.2 *Let P be a program:*

$$\begin{aligned} a &\leftarrow \text{even}(\{a, b, c\}) \\ b &\leftarrow \text{even}(\{a, b, c\}) \\ c &\leftarrow \text{even}(\{a, b, c\}) \end{aligned}$$

If we try to apply the one-step provability operator to this program, we get the sequence:

$$\begin{aligned} T_P(\emptyset) &= \{a, b, c\} \\ T_P(\{a, b, c\}) &= \emptyset \\ &\vdots \end{aligned}$$

The T_P operator does not stabilize to a fixpoint but instead oscillates between two values.

The essential difference between $\text{Card}(b, S)$ and $\text{even}(S)$ is that $\text{Card}/2$ is monotone while $\text{even}/1$ is not: if $M \models \text{Card}(b, S)$, then for all sets

$M' \supseteq M$ it holds that $M' \models \text{Card}(b, S)$. We saw in the previous example that this does not hold for parity aggregates.

If we want to allow non-monotone aggregates, we have to define their semantics in a different way. The standard approach [107] is to treat them like negative literals: when reducing the program we replace all satisfied aggregates by \top and unsatisfied by \perp . However, in this case we have to be careful with the definitions since otherwise we may end with a situation where an atom justifies itself via an aggregate. For example, if we have the rule:

$$a \leftarrow \text{Card}(1, \{a\}) \quad (7.1)$$

then its reduct with respect to $M_1 = \{a\}$ is

$$a \leftarrow \top$$

with the least model $\{a\}$ — a is true because a is true. This problem can be avoided by admitting only set-inclusion minimal justified models as answer sets [21, 42, 69]. In the above rule $M_2 = \emptyset$ is also a stable model and $M_2 \subset M_1$ so we reject M_1 .

This approach causes a problem with choice rules because choice rules produce answer sets that are not minimal. For example, if we added a new rule to the previous example to get:

$$\begin{aligned} a &\leftarrow \text{Card}(1, \{a\}) \\ \{b\} &\leftarrow \end{aligned}$$

we still have only one minimal answer set: \emptyset . In effect, we lost the answer set $\{b\}$.

Another approach for nonmonotonic aggregates is presented by Son et. al. [183, 182] where the semantics is not defined in terms of reducts but instead the provability operator T_P is defined using the concept of conditional satisfaction. A set of atoms R satisfies an aggregate A conditionally with respect to a set of atoms S if R satisfies A and all sets R' , $R \subseteq R' \subseteq S$ satisfy A . The one-step provability operator is then defined to have two arguments, R and S and it derives a new atom if the body is conditionally satisfied by R with respect to S . Including the set S in T_P removes the need for constructing an explicit reduct for it.

When we apply this semantics to (7.1), we find that:

$$\begin{aligned} T_P(\emptyset, \emptyset) &= \emptyset \\ T_P(\emptyset, \{a\}) &= \emptyset \end{aligned}$$

so \emptyset is the only answer set. Even though Son et. al. allow a rule to have an arbitrary aggregate in the head, they restrict T_P so that only basic rules can generate atoms to models. Thus, this approach as presented in [183] is also incompatible with choice rules.

Marek et. al. [126] define semantics for monotone aggregates by using a non-deterministic provability operator where T_P selects one set of atoms that satisfies the head of the rule and makes them true. This approach allows the use of monotone aggregates in the rule heads without causing any problems. We could have used this approach for defining the semantics of cardinality constraint logic programs but we wanted to keep the basic language as simple as possible in this work.

7.1.7 On Function Symbols

Marek et al. [134] showed that the stable model semantics of normal logic programs is undecidable when function symbols are allowed.³ There has been some work in identifying classes of programs where at least some questions are decidable. One such a formalism is finitary programs of Bonatti [11].

A normal logic program is finitary if and only if:

1. every ground atom in its Herbrand instantiation depends only on a finite number of atoms; and
2. only a finite number of ground atoms take part in odd loops.⁴

Bonatti proves that given such a program P and a ground atom A , the question whether A is true in some stable model of P (or in all stable models) is decidable.

To see this note that the question whether a normal program has a stable model at all can be decided by examining the odd loops of the program [70]. If there is only a finite number of atoms participating in such loops, we can systematically check through all possible truth value combinations for them to see if we have one or more stable models. Next, if an atom A depends only on a finite number of atoms, we can check whether it can be deduced by examining only rules for those atoms.

A relevant instantiation of a finitary program P with respect to an atom A contains all rules for A and the atoms that it depends on as well as all instances of odd loops. By definition the instantiation is finite so we can compute its stable models and see if A is true.

The finitary programs are closer to conventional logic programming languages than to most ASP languages. It is possible that a finitary program has an infinite stable model. For example, the program:

$$\begin{aligned} p(0) &\leftarrow \\ p(s(X)) &\leftarrow p(X) \end{aligned}$$

is finitary but has a unique infinite stable model. This means that with finitary programs we have to write programs so that we can guarantee that the interesting part of the stable model is finite. On the other hand, ω -restricted programs have guaranteed finite stable models.

Several questions about finitary programs are undecidable [11]. For example, the question whether some instance of a non-ground atom A is true is semi-decidable. Also, the question whether a normal logic program is finitary or not is undecidable in the general case. Bonatti identifies a syntactic class of programs that are finitary.

³In fact, they showed that when negative literals are allowed, the stable model semantics spans the whole arithmetic hierarchy [40].

⁴An odd loop is a negative loop that contains an odd number of atoms in the dependency graph of the Herbrand instantiation of a program.

7.2 THE FULL LANGUAGE SYNTAX

We now introduce the syntax of the full language. We use the same classes of syntactic elements as we did for basic programs in Chapter 2. We will elaborate on the nature of the new elements when we introduce translations for them.

Terms

A *term* is either a *variable*, a *function term*, or a *range* $t_1 \dots t_2$ where t_1 and t_2 are ground terms as defined in the basic language.

The set of function symbols includes the arithmetic operators from the standard interpretation⁵ as well as a function symbol *norm*/1 and possibly a set of *command line constants* that are constants that are interpreted as numbers.

Literals

A *basic literal* is either an atom A or its negation $\text{not}(A)$. A *conditional literal* \mathcal{L} is of the form:

$$X.L : A_1 \wedge \dots \wedge A_n$$

where the *main literal* L is a basic literal, the A_i in the *condition* are atoms, and X is a set of *local variables*.

A *cardinality atom* C is either $\text{Card}(l, S)$ where the bound l is a term that is not a range or $\text{Card}_u(l, u, S)$ where the *lower bound* l and the *upper bound* u are terms that are not ranges and S is a set of conditional literals. The intuition is now that the number of true literals in S has to be between l and u , inclusive. A *cardinality literal* \mathcal{C} is either a positive cardinality atom C or a negative cardinality literal $\text{not } C$.

Rules and Programs

A *rule* is either a basic rule, a choice rule, or an *extended* rule of the form:

$$\mathcal{C}_0 \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$$

where the head \mathcal{C}_0 is a cardinality atom and \mathcal{C}_i in the body are cardinality literals. A *full-language cardinality constraint program* (FCCP) is a set of rules.

Syntactic Sugar

A basic literal L that occurs by itself in a rule body is interpreted as a shorter way to write the cardinality atom $\text{Card}(1, \{L : \top\})$.

A conditional literal $X.L : A$ may occur in a rule body by itself if $\text{Var}(A) \subseteq X$ and then it is a shortcut for the cardinality atom

$$\text{Card}(\text{norm}(\text{pred}(A)), \{X.L : A\}) .$$

The interpretation of *norm* will be defined later in Section 7.3.8.

For example, the rule:

$$\text{maximum}(X) \leftarrow \{Y\}.\text{less-equal}(Y, X) : d(Y), d(X)$$

⁵Table 3.1 on page 40.

is a notational shortcut for the rule:

$$\begin{aligned} \text{maximum}(X) \leftarrow & \text{Card}(\text{norm}(d/1), \{Y.\text{less-equal}(Y, X) : d(Y)\}), \\ & \text{Card}(1, \{d(X) : \top\}) . \end{aligned}$$

7.3 TRANSFORMATIONS

We define the semantics of the full language by defining a translation from the full language into basic programs. The idea is that we define a number of functions that each transform one feature of the full language into a corresponding basic construct, and then we compose all the partial transformations into one.

7.3.1 On Notations

We use the notations CCP and FCCP to denote the whole classes of basic and full language programs, respectively.

For rules we will use the variant notation where $\langle H, \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \rangle$ stands for the rule $H \leftarrow \mathcal{C}_1, \dots, \mathcal{C}_n$ in the transformations.

The transformation that we are going to define is denoted with \mathcal{T} and it is a function $\mathcal{T} : \text{FCCP} \rightarrow \text{CCP}$. It is constructed as a composition of component transformations that are functions $\mathcal{T}^c : \text{FCCP} \rightarrow \text{FCCP}$ that produce a program that contains only those features that are present in the basic language followed by a trivial mapping $\mathcal{T}^i : \text{FCCP} \rightarrow \text{CCP}$ that just interprets the program as a basic program.

When we define the component transformations, we use similar notations to define auxiliary constructs. If a function \mathcal{T}^c transforms the full language feature c , then \mathcal{T}_R^c is an auxiliary that translates a single rule, \mathcal{T}_C^c is an auxiliary for handling cardinality literals, and \mathcal{T}_L^c is for conditional literals.

We will need an auxiliary notation for creating an atom that has a sequence of variables as its arguments.

Definition 7.3.1 *If $V = \langle V_1, \dots, V_n \rangle$ is a sequence of variables and p is an n -ary predicate symbol, then $p(\overline{V})$ denotes the atom $p(V_1, \dots, V_n)$.*

For example, if $V = \langle X, Y \rangle$, then $p(\overline{V})$ is the atom $p(X, Y)$. In case we have a set of variables, then we start by putting the variables in the lexicographic order. Thus, if $S = \{X, Y, A\}$, then $p(\overline{S}) = p(A, X, Y)$.

7.3.2 Upper and Lower Bounds for Cardinality Atoms

It is useful to be able to state also the upper limit for the number of literals that are true in cardinality atoms. For example, in a planning problem we might want to express a condition that a taxi may have between one to four passengers. One way to model this would is:

$$\leftarrow \text{not } 1 \{P.\text{passenger}(T, P) : \text{person}(P)\} 4, \text{taxi}(T) .$$

We can remove the upper bound from a cardinality literal by replacing it by a pair of literals with only a lower bound. However, we have to handle positive and negative literals separately. This is because a positive literal is expressing a conjunction (the number is at least lower bound *and* at most upper bound) while a negative one expresses a disjunction (the number is either smaller than the lower bound *or* greater than the upper bound). In the positive case we put both of the new literals to the same rule body, but in the negative case we have to put them into different rules.

If $\text{Card}_u(l, u, S)$ is a positive cardinality literal with both lower and upper bounds, we can represent it with a pair of cardinality literals:

$$\begin{aligned} & \text{Card}(l, S) \\ & \text{not Card}(u + 1, S). \end{aligned}$$

For example, the rule:

$$ok \leftarrow 1 \{X.chosen(X) : option(X)\} 3$$

is translated to:

$$ok \leftarrow 1 \{X.chosen(X) : option(X)\}, \text{not } 4 \{X.chosen(X) : option(X)\} .$$

We translate a negative cardinality literal $\text{not Card}_u(l, u, S)$

$$\begin{aligned} & \text{not Card}(l, S) \\ & \text{Card}(u + 1, S) . \end{aligned}$$

In this case we take two copies of the original rule and place one new cardinality literal to both. For our previous example we get two rules:

$$\begin{aligned} & \leftarrow \text{not } 1 \{P.passenger(T, P) : person(P)\}, taxi(T) \\ & \leftarrow 5 \{P.passenger(T, P) : person(P)\}, taxi(T) . \end{aligned}$$

When we have n negative cardinality atoms with both bounds in a rule, we have to create 2^n copies of the rule, one for every possible combination. While this may result in a rather large explosion on the size of the program, it is not a serious problem in practice since we usually have only one or at most two cardinality literals that are not equivalent to basic literals in a rule body.

Definition 7.3.2 *The function $\mathcal{T}^u: FCCP \rightarrow FCCP$ is defined as:*

$$\mathcal{T}^u(P) = \bigcup_{R \in P} \mathcal{T}_R^u(R)$$

where $\mathcal{T}_R^u(R)$ is defined as:

$$\mathcal{T}_R^u(\langle H, \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \rangle) = \{ \langle H, \{\mathcal{C}'_1, \dots, \mathcal{C}'_n\} \rangle \mid \mathcal{C}'_i \in \mathcal{T}_{\mathcal{C}_i}^u(\mathcal{C}_i) \}$$

and $\mathcal{T}_{\mathcal{C}}^u(\mathcal{C})$ is defined as:

$$\mathcal{T}_{\mathcal{C}}^u(\mathcal{C}) = \begin{cases} \{ \{ \text{Card}(l, S), \text{not Card}(u + 1, S) \} \}, & \text{if } \mathcal{C} = \text{Card}_u(l, u, S) \\ \{ \{ \text{not Card}(l, S) \}, \{ \text{Card}(u + 1, S) \} \}, & \text{if } \mathcal{C} = \text{not Card}_u(l, u, S) \\ \{ \mathcal{C} \}, & \text{otherwise} . \end{cases}$$

The final case corresponds to the case where we have ordinary cardinality literals with only lower bounds.

There is an alternative translation for removing the upper bounds that we can use if we are willing to relax the condition that a translation should not introduce new atoms into the non-domain Herbrand base of the program. Consider again the rule:

$$ok \leftarrow 1 \{X.chosen(X) : option(X)\} 3$$

We add a new rule for the upper bound to get the alternative translation:

$$\begin{aligned} ok &\leftarrow 1 \{X.chosen(X) : option(X)\}, \text{not } p \\ p &\leftarrow 4 \{X.chosen(X) : option(X)\} \end{aligned}$$

where p is a new atom. If the cardinality atom has any global variables in it, we add them as the arguments of the new atom. In this case we have to add also the domain literals to the body of the new rule. For example,

$$q(Y) \leftarrow 1 \{X.r(X, Y) : d(X)\} 2, d(Y)$$

is translated to:

$$\begin{aligned} q(Y) &\leftarrow 1 \{X.r(X, Y) : d(X)\}, \text{not } p(Y), d(Y) \\ p(Y) &\leftarrow 3 \{X.r(X, Y) : d(X)\}, d(Y) . \end{aligned}$$

This translation avoids the exponential worst-case size increase of Definition 7.3.2 but the cost is that we get a new ground atom into the non-domain Herbrand base for every ground instance of every upper bound that occurs in a rule.

7.3.3 Full Cardinality Atoms in Rule Heads

In many cases we would like to have cardinality atoms in the rule heads. For example, the condition of the Hamilton cycle problem that each vertex in a graph has to have exactly one outgoing edge in the cycle can be expressed as:

$$1 \{Y.hc(X, Y) : edge(X, Y)\} 1 \leftarrow vtx(X) .$$

We translate this to two rules, one choice rule and a constraint:

$$\begin{aligned} \{hc(X, Y)\} &\leftarrow edge(X, Y), vtx(X) . \\ \leftarrow \text{not } 1 \{Y.hc(X, Y) : edge(X, Y)\} 1, vtx(X) . \end{aligned}$$

If we have more than one conditional literal in the head of the rule, we create a new choice rule for each one.

The formal definition of this transformation is straightforward.

Definition 7.3.3 *The function $\mathcal{T}_h: FCCP \rightarrow FCCP$ is defined as:*

$$\mathcal{T}^h(P) = \bigcup_{R \in P} \mathcal{T}_R^h(R)$$

where $\mathcal{T}_R^h(R)$ of an extended rule $R = \langle C, \text{body} \rangle$ with a cardinality atom $C = \text{Card}_u(b, u, \{\mathcal{L}_1, \dots, \mathcal{L}_n\})$ as its head is:

$$\mathcal{T}_R^h(R) = \{ \langle \{L\}, \{A\} \cup \text{body} \rangle \mid L : A \in C \} \\ \cup \langle \perp, \{\text{not } C\} \cup \text{body} \rangle .$$

The $\mathcal{T}_R^h(R)$ of an ordinary rule $R = \langle H, \text{body} \rangle$ is the identity function $\mathcal{T}_R^h(R) = \langle H, \text{body} \rangle$.

7.3.4 Multiple Conditions in Conditional Literals

it is often useful to have more than one atom as the condition of a conditional literal. This is particularly useful if we have built-in predicates that define basic relations such as equality or less-than. We can translate these by adding a new atom that combines all the conditions in its definition. For example, if we have a conditional literal:

$$Y.\text{choose}(X, Y) : \text{option}(Y) \wedge \text{greater}(Y, 10)$$

we translate it to:

$$Y.\text{choose}(X, Y) : d(Y)$$

and a new rule:

$$d(Y) \leftarrow \text{option}(Y), \text{greater}(Y, 10) .$$

In case of ω -restricted programs there is still one consideration that we have to address: If the conditions contain global variables, we need to add domain literals for them to the rule body, too. For example, if we have the rule:

$$\text{ok}(X) \leftarrow 2 \{ Y.\text{chosen}(X, Y) : \text{option}(X, Y) \wedge \text{greater}(Y, Z) \}, \\ \text{choice}(X), \text{limit}(Z)$$

we have to add also the atoms $\text{limit}(Z)$ and $\text{choice}(X)$ to the body of the new rule to get:

$$\text{ok}(X) \leftarrow 2 \{ Y.\text{chosen}(X, Y) : d(X, Y, Z) \}, \\ \text{choice}(X), \text{limit}(Z) \\ d(X, Y, Z) \leftarrow \text{option}(X, Y), \text{greater}(Y, 10), \text{limit}(Z), \text{choice}(X) .$$

Note that $d/3$ is a domain predicate. Since a domain predicate has a fixed extension, adding one does not introduce new choices to the program so it is computationally cheaper than adding a new non-domain predicate symbol.

Definition 7.3.4 Let $\mathcal{L} = X.L : A_1 \wedge \dots \wedge A_n$ where $n > 1$ be a conditional literal, $V = \bigcup_{i \in [1, n]} \text{Var}(A_i)$ be the set of variables occurring in the condition, and $d_{\mathcal{L}}$ be a $|V|$ -ary predicate symbol. Then

$$\mathcal{T}_{\mathcal{L}}^{\wedge}(\mathcal{L}) = X.L : d_{\mathcal{L}}(\overline{V}) \\ S_{\mathcal{L}}(\mathcal{L}) = \{ \langle d_{\mathcal{L}}(\overline{V}), \{A_1, \dots, A_n\} \rangle \} .$$

When $\mathcal{L} = X.L : A$, $\mathcal{T}_\mathcal{L}^\wedge(\mathcal{L}) = \mathcal{L}$ and $S_\mathcal{L}(\mathcal{L}) = \emptyset$.

Let $C = \text{Card}(b, S)$ be a cardinality atom. Then,

$$\begin{aligned}\mathcal{T}_C^\wedge(C) &= \text{Card}(b, \{\mathcal{T}_\mathcal{L}^\wedge(\mathcal{L}) \mid \mathcal{L} \in S\}) \\ S_C(C) &= \bigcup_{\mathcal{L} \in S} S_\mathcal{L}(\mathcal{L})\end{aligned}$$

Let $R = \langle C_0, \{C_1, \dots, C_n\} \rangle$ be a rule. Then,

$$\begin{aligned}\mathcal{T}_r^\wedge(R) &= \{\langle \mathcal{T}_C^\wedge(C_0), \{\mathcal{T}_C^\wedge(C_1), \dots, \mathcal{T}_C^\wedge(C_n) \} \rangle\} \\ &\cup \{ \langle H, B \cup \text{body}_\mathcal{D}(R) \rangle \mid \langle H, B \rangle \in \bigcup_{i \in [0, n]} S_C(C_i) \}\end{aligned}$$

The transformation $\mathcal{T}^\wedge : FCCP \rightarrow FCCP$ is defined as:

$$\mathcal{T}^\wedge(P) = \bigcup_{R \in P} \mathcal{T}_r^\wedge(R) .$$

7.3.5 Conditional Literals in Rule Bodies

We can use a conditional literal to express existential quantification. For example, in many contexts we can interpret $\text{Card}(1, \{X.a(X) : b(X)\})$ as saying “there exists X such that $a(X)$ and $b(X)$ are true”. In the full language we allow those literals to occur in rule bodies by themselves and we can express the universal quantification with them.

For example, we can express the statement that X is the maximum of a total order with the rule:

$$\text{maximum}(X) \leftarrow \{Y\}.less\text{-equal}(Y, X) : d(Y), d(X) .$$

Earlier, we defined this to be equivalent to

$$\begin{aligned}\text{maximum}(X) &\leftarrow \text{Card}(\text{norm}(d/1), \{Y.less\text{-equal}(Y, X) : d(Y)\}), \\ &\text{Card}(1, \{d(X) : \top\}) .\end{aligned}$$

Supposing that our elements are natural numbers from zero to two, we get the ground instances:⁶

$$\begin{aligned}\text{maximum}(0) &\leftarrow \text{Card}(3, \{less\text{-equal}(0, 0), less\text{-equal}(1, 0), less\text{-equal}(2, 0)\}) \\ \text{maximum}(1) &\leftarrow \text{Card}(3, \{less\text{-equal}(0, 1), less\text{-equal}(1, 1), less\text{-equal}(2, 1)\}) \\ \text{maximum}(2) &\leftarrow \text{Card}(3, \{less\text{-equal}(0, 2), less\text{-equal}(1, 2), less\text{-equal}(2, 2)\})\end{aligned}$$

Supposing that we use the standard definition for less-than-or-equal, only the last rule has a true body and $\text{maximum}(2)$ is the only atom that we can derive with them.

When defining this construct (p. 118) we limited ourselves to conditional literals that have only one condition and we had the additional limitation that all variables in it have to be local. The reason for these limitations is technical: the *norm* function takes as its input a domain

⁶With the domain predicate left out for clarity.

predicate symbol and it returns the size of its extension. If there is more than one condition, then we do not have a unique predicate symbol anymore to give us the bound for the literal. If the condition contains global variables, then the bound that we get with *norm* may be too large.

These limitations are not essential. We imposed them so that we did not have to worry about special cases in the definition of \mathcal{T} . We can add both multiple conditions and global variables with the construct that we used in defining \mathcal{T}^\wedge in Section 7.3.4.

There is one alluring candidate translation where we could escape this limitation directly. We could say that $X.a(X) : d(X)$ denotes the negative cardinality literal not 1 $\{\text{not } a(X) : d(X)\}$. This translation corresponds to the classical equivalence of $\forall x.p(X)$ and $\neg\exists x\neg p(x)$. Under classical logic it would work flawlessly, but it has an undesirable side effect under the stable model semantics: the double negation can be used to justify atoms. For example, the program:

$$a \leftarrow \text{not } 1 \{\text{not } a : \top\}$$

has two stable models $\{a\}$ and \emptyset . This means that under this interpretation we could not use the construct to express the condition “ $a(y)$ is true if $\forall x.p(x, y)$ holds.”

7.3.6 Integral Ranges

An integral range has the form $l..u$ where l and u are terms that are interpreted as integers. They are mostly used to define a large number of facts with one rule. For example, the fact:

$$\text{number}(1..k) \leftarrow$$

corresponds to the set of facts:

$$\begin{array}{l} \text{number}(1) \leftarrow \\ \vdots \\ \text{number}(k) \leftarrow . \end{array}$$

The behavior of a range depends on whether it is used in the head or in the body of a rule. In the head the natural interpretation is that it represents a disjunction while in the body a conjunction is more intuitive meaning. For example,

$$h \leftarrow 2 \{a(1..k)\}$$

corresponds to

$$h \leftarrow 2 \{a(1), a(2), \dots, a(k)\} .$$

In the translation we replace a range by a new variable and a new unary domain predicate, and then define facts for that predicate.

For example, if we have a rule:

$$\text{grid}(1..5, 1..3) \leftarrow ,$$

we replace it by

$$\text{grid}(X, Y) \leftarrow d_x(X), d_y(Y)$$

and the facts defining the atoms $d_x(1), \dots, d_x(5), d_y(1), \dots, d_y(3)$.

As in the first example, at least one of the bounds is usually an interpreted constant. This means that we cannot do the actual transformation until we know its interpretation so we have to do this transformation when we instantiate the program.

In the formal definition we construct the transformation from bottom-up, starting at the level of ranges themselves. The notations \mathcal{T}^r denote the transformed expressions, the sets V contain the new variables, and the sets D contain the new domain literals we need.

Definition 7.3.5 *Let t be a term. Then, \mathcal{T}_t^r is defined as:*

$$\mathcal{T}_t^r(t) = \begin{cases} V_t, & t = l..u \text{ and } V_t \text{ is a new variable} \\ t, & \text{otherwise} \end{cases}.$$

Let $A = p(t_1, \dots, t_n)$ be an atom. Then, \mathcal{T}_A^r is defined as:

$$\mathcal{T}_A^r(A) = p(\mathcal{T}^r(t_1), \dots, \mathcal{T}^r(t_n))$$

and the sets $V(A)$ and $D(A)$ are defined as follows:

$$\begin{aligned} V(A) &= \{V_{t_i} \mid t_i = l..u, 1 \leq i \leq n\} \\ D(A) &= \{d_{t_i}(V_{t_i}) \mid t_i = l..u, 1 \leq i \leq n\} \end{aligned}.$$

For a negative basic literal $L = \text{not } A$, $\mathcal{T}_L^r(L) = \text{not } \mathcal{T}_A^r(A)$, $V(L) = V(A)$, and $D(L) = D(A)$. Let $\mathcal{L} = X.L : A$ be a conditional literal. Then, $V(\mathcal{L}) = V(L) \cup V(A)$, $D(\mathcal{L}) = D(L) \cup D(A)$, $\mathcal{T}_{\mathcal{L}}^r(\mathcal{L}) = X \cup V(\mathcal{L}) \cup D(A) \cdot \mathcal{T}_L^r(L) : \mathcal{T}_A^r(A) \wedge (\bigwedge \{A \in D(\mathcal{L})\})$.

Let $C = \text{Card}(b, S)$ be a cardinality atom, then we have

$$\mathcal{T}_C^r(C) = \text{Card}(b, \{\mathcal{T}_{\mathcal{L}}^r(\mathcal{L}) : \mathcal{L} \in S\})$$

and for a negative cardinality literal $\mathcal{C} = \text{not } C$, $\mathcal{T}_{\mathcal{C}}^r(\mathcal{C}) = \text{not } \mathcal{T}_C^r(C)$.

For a rule $R = \langle H, \{\mathcal{C}_1, \dots, \mathcal{C}_n\} \rangle$,

$$\mathcal{T}_R^r(R) = \langle \mathcal{T}_A^r(H), \{\mathcal{T}_{\mathcal{C}_1}^r(\mathcal{C}_1), \dots, \mathcal{T}_{\mathcal{C}_n}^r(\mathcal{C}_n)\} \cup D(H) \rangle.$$

Let the transform $\mathcal{T}^r : FCCP \rightarrow FCCP$ be the function:

$$\mathcal{T}^r(P) = \bigcup_{R \in P} \mathcal{T}^r(R) \cup F_r(D_R)$$

where

$$\begin{aligned} F_r(D_R) &= \{ \langle d_{t_i}(n), \emptyset \rangle \mid d_{t_i}(V_{p_i}) \in D_R(R), t_i = l..u, \\ &\quad \text{and } I(l) \leq n \leq I(u) \} \end{aligned}.$$

Example 7.3.1 *Consider the rule:*

$$a(1..3, X) \leftarrow 2 \{Y.b(Y, 1..2) : d(Y)\}, d(X).$$

On the atom level the transformations for the two ranges are:

$$\begin{aligned} \mathcal{T}_A^r(a(1..3, X)) &= a(V_{1..3}, X) \\ \mathcal{T}_A^r(b(Y, 1..2)) &= b(Y, V_{1..2}) \end{aligned}$$

The sets of new variables and domain literals are:

$$\begin{aligned} V(a(1..3, X)) &= \{V_{1..3}\} \\ V(b(Y, 1..2)) &= \{V_{1..2}\} \\ D(a(1..3, X)) &= \{d_{1..3}(V_{1..3})\} \\ V(b(Y, 1..2)) &= \{d_{1..2}(V_{1..2})\} . \end{aligned}$$

The complete transformation $\mathcal{T}_R^r(R)$ of the rule is:

$$\begin{aligned} a(V_{1..3}, X) \leftarrow & 2 \{ \{Y, V_{1..2}\}.b(Y, V_{1..2}) : d(Y) \wedge d_{1..2}(V_{1..2}) \}, \\ & d(X), d_{1..3}(V_{1..3}) \\ d_{1..3}(1) \leftarrow & \\ d_{1..3}(2) \leftarrow & \\ d_{1..3}(3) \leftarrow & \\ d_{1..2}(1) \leftarrow & \\ d_{1..2}(2) \leftarrow & . \end{aligned}$$

Integral ranges do not fit nicely into our two-level language definition but they are too useful in practice to be left out. Almost all of our problem encodings in Chapters 9 and 10 use ranges to define some of the domain predicates.

The problem is that whenever we have interpreted constants in range definitions, we cannot transform them until we know what values those constants take and we do not find that out until we are handling actual problem instances. We could escape the problem by introducing ranges directly into the basic language but that would complicate its definition.

Luckily, the problem is largely cosmetic as the semantics of ranges is intuitive and the transformation itself is simple enough that it does not cause any practical problems if we defer transforming them until immediately before the instantiation.

However, this causes another problem since we added the new domains in the rule body were as conditions for the conditional literals. This means that we have to remove them. We usually do the removing multiple conditions as the first step of the translation as we need to get them out of rule heads before we can process the heads further. Luckily, multiple conditions in rule bodies do not cause any semantic trouble so we just wrap the $\mathcal{T}_R^r(r)$ operations inside a $\mathcal{T}^\wedge(R)$ operation that completes the translation.

7.3.7 The Complete Transformation

The transformation from the full language to the basic language is defined as a large composition of the previously defined component parts.

We do the composition in the order that we

1. remove multiple conditions (\mathcal{T}^\wedge);
2. remove cardinality atoms from rule heads (\mathcal{T}^h);
3. remove upper bounds from cardinality atoms (\mathcal{T}^u); and

4. remove ranges (\mathcal{T}^r); and

After doing these translations we are left with a full language program that contains only elements that are present in basic programs. Then, we just add a trivial partial function $\mathcal{T}^i : \text{FCCP} \rightarrow \text{CCP}$ where $\mathcal{T}^i(P) = P$ for all P that are CCPs and that is undefined for all other programs.

Definition 7.3.6 *Let $\mathcal{T} : \text{FCCP} \rightarrow \text{CCP}$ be the function:*

$$\mathcal{T}(P) = \mathcal{T}^r(\mathcal{T}^u(\mathcal{T}^h(\mathcal{T}^\wedge(P)))) .$$

Here the order of the functions in the composition is relevant. For example, \mathcal{T}^u that removes upper bounds from cardinality literals assumes that \mathcal{T}^h has been already applied to remove any cardinality atoms that occur in the head of some rule.

7.3.8 Augmenting Standard Interpretation

We will augment the standard interpretation (Section 3.9) with another interpreted function symbol, $\text{norm}/1$. This function takes as its sole argument a domain predicate symbol, and it then returns the size of its extension in the domain model.

For example, consider the program.

$$\begin{aligned} p(1) &\leftarrow \\ p(2) &\leftarrow \\ q(\text{norm}(p)) &\leftarrow \end{aligned}$$

In this example $\text{norm}(p) = 2$ since the extension of p in the domain model is $\{p(1), p(2)\}$. Thus, the program has the stable model $\{p(1), p(2), q(2)\}$.

The definition of $\text{norm}/1$ has the same form as the definitions presented in Section 3.9.

Definition 7.3.7 *The interpretation function $I_{||} : U \rightarrow U$ is defined as:*

$$I_{||}(t) = \begin{cases} n, & \text{if there is a predicate symbol } t \in \mathcal{P}_D(P) \text{ and the size} \\ & \text{of the extension of } t \text{ is } n \text{ in the domain model } D_\omega \text{ of } P; \\ \mathbf{e}, & \text{otherwise} . \end{cases}$$

7.4 FURTHER EXTENSIONS

In this section we present two further extensions to the language. The first one is a weight literal that generalizes the notion of a cardinality atom. The second one is classical negation that introduces a new way of handling negations in programs.

7.4.1 Weight Literals and Rules

The cardinality atoms may be generalized to *weight atoms* [179]. There every literal in the atom has a weight defined for it and the atom is

satisfied if the sum of the weights of the satisfied literals exceeds the bound. We use the syntax:

$$\text{Weight}(b, S)$$

where b is again an integral bound and $S = \{\mathcal{L}_1 = w_1, \dots, \mathcal{L}_n = w_n\}$ where w_i are non-negative integral weights of the literals \mathcal{L}_i . We also use the notation $w(\mathcal{L}_i) = w_i$. The alternate SMOELS syntax for a weight atom is:

$$b [\mathcal{L}_1 = w_1, \dots, \mathcal{L}_n = w_n] .$$

Let S be any set of weighted literals. Then, we use $w(S)$ to denote the sum of weights of literals in S :

$$w(S) = \sum_{\mathcal{L} \in S} w(\mathcal{L}) .$$

A weight atom is satisfied when the sum of the weights of satisfied literals exceeds the bound:

$$M \models \text{Weight}(b, S) \text{ if and only if } w(\{\mathcal{L} \in S \mid M \models \mathcal{L}\}) \geq b .$$

The reducts for weighted conditional literals and weight atoms are defined analogously to the unweighted ones.

Definition 7.4.1 (Modified from Def. 3.3.1) *Let $\mathcal{L} = L : A = w$ be a weighted conditional literal. Then, its reduct \mathcal{L}^M respect to the set of atoms M is:*

$$\mathcal{L}^M = \begin{cases} \{L : A = w\}, & \text{if } L \text{ is positive} \\ \{\top : A = w\}, & \text{if } L \text{ is negative and } M \models L \\ \emptyset, & \text{if } L \text{ is negative and } M \not\models L . \end{cases}$$

Let $\mathcal{W} = \text{Weight}(b, S)$ be a weight atom. Then its reduct \mathcal{W}^M with respect to the set of atoms M is:

$$\mathcal{W}^M = \text{Weight}(b, S^M)$$

where

$$S^M = \bigcup_{\mathcal{L} \in S} \mathcal{L}^M .$$

When we further extend weight atoms to allow the use of variables in both bounds and weights itself, we get even more expressivity. For example, we can use them to get a uniform encoding for the KNAPSACK problem that is tricky to do in ASP without weights.

Example 7.4.1 **PROBLEM 4: KNAPSACK.** *Given a set of objects $O = \langle o_1, \dots, o_n \rangle$ that each have a weight w_i and a value v_i attached to them and two integers c and t , decide whether there exists a subset $K \subseteq O$ of objects such that the sum of values of objects in K is at least t while their total weight is at most c .*

This problem can be solved with the program:

$$\begin{aligned} & \{\{X, W, V\}.in(X) : object(X, W, V)\} \leftarrow \\ & \quad \leftarrow C + 1 [\{X, W, V\}.in(X) : object(X, W, V) = W], capacity(C) \\ & \quad \leftarrow [\{X, W, V\}.in(X) : object(X, W, V) = V] T - 1, target(T) . \end{aligned}$$

An atom $object(x, w, v)$ denotes that x is an object with a weight of w and a value of v . The first rule gives us a free choice over all objects, and the second one weeds out all those model candidates that exceed the carrying capacity of the knapsack. The final rule rejects those candidates where the value is not great enough.

In this work we consider only weights that are not negative. This is because negative weights add a new source of non-monotonicity to the programs. For example, consider the weight atom $W = \text{Weight}(1, \{a = 1, b = -1\})$. Now, $\{a\} \models W$ but $\{a, b\} \not\models W$ even though all literals in it are positive.

Niemelä et.al. [147, 145] eliminated negative weights from weight literals by making the observation that a negative weight is essentially a penalty for having that literal true. If an atom A carries a penalty of $-w$ with it, we can turn it around and say that not A has the bonus of w and by increasing the bounds by the same amount. However, Ferraris and Lifschitz [72] have identified situations where this approach leads to unintuitive results.

Example 7.4.2 *Consider the program P_1 :*

$$\begin{aligned} a & \leftarrow 1 [b = 0] \\ b & \leftarrow 1 [a = 0] . \end{aligned}$$

This program has only one stable model, $M = \emptyset$. If we lower both the bounds and the weights by one, we would expect to get a program equivalent to this one. However, the program P_2 :

$$\begin{aligned} a & \leftarrow 0 [b = -1] \\ b & \leftarrow 0 [a = -1] \end{aligned}$$

is translated into P'_2 :

$$\begin{aligned} a & \leftarrow 1 [\text{not } b = 1] \\ b & \leftarrow 1 [\text{not } a = 1] . \end{aligned}$$

This program has two stable models, $M_1 = \{a\}$ and $M_2 = \{b\}$.

The main reason for this unintuitive behavior is that turning a positive literal into a negative one potentially introduces a choice point to the program.

7.4.2 Strong Negation

The standard form of negation that is used in answer set programming is the default negation: we assume by default that the a literal not A is true if we cannot show that A is true. The second type of negation is called *strong negation*⁷: a literal $\neg A$ is true if we can prove that A is

⁷Strong negation is also called *classical negation*.

false.

One example that shows the conceptual difference between the default and strong negations is the railroad crossing problem [85]. We want to cross the tracks only if a train is not coming. A naive way to model it using default negation would use rules of the form:

$$\begin{aligned} \textit{cross-tracks} &\leftarrow \text{not } \textit{train-coming} \\ \textit{train-coming} &\leftarrow \textit{observe-train}, \text{not } \textit{empty-tracks} . \end{aligned}$$

The problem here is that $\text{not } \textit{train-coming}$ is true whenever $\textit{observe-train}$ is false. If we fail to do the observation for some reason, we conclude that it is safe to cross the tracks.

If we use the strong negation:

$$\textit{cross-tracks} \leftarrow \neg \textit{train-coming}$$

we have to explicitly prove that $\neg \textit{train-coming}$ is true before crossing. For example, we might have a rule:

$$\neg \textit{train-coming} \leftarrow \textit{observe-train}, \textit{empty-tracks} .$$

We will use strong negation in several examples in Chapter 10.

We will implement a strong negation $\neg A$ by defining a new atom A' to act as the negation of the A and then adding a rule to forbid both A and A' from being true. For example, the previous train crossing example becomes:

$$\begin{aligned} \textit{cross-tracks} &\leftarrow \textit{train-coming}' \\ \textit{train-coming}' &\leftarrow \textit{observe-train}, \textit{empty-tracks} \\ &\leftarrow \textit{train-coming}, \textit{train-coming}' . \end{aligned}$$

When a program has variables, we work with the same principle, but we have to also handle the domain literals that occur in rule bodies. For example, we would translate the rule:

$$\neg p(X) \leftarrow \text{not } p(X), d(X)$$

into

$$\begin{aligned} p'(X) &\leftarrow \text{not } p(X), d(X) \\ &\leftarrow p(X), p'(X), d(X) . \end{aligned}$$

We do this for every rule that has a strong negation in the head.

This approach differs from the original definition of strong negation in the context of stable model semantics. Gelfond and Lifschitz [85] allow a program to have a unique inconsistent stable model that contains all atoms if the program is inconsistent. For example, the program:

$$\begin{aligned} a &\leftarrow \\ \neg a &\leftarrow \end{aligned}$$

has a unique stable model $\{a, \neg a\}$ under semantics of [85] but it does not have a stable model under the above translation.

The reason for this difference is mainly pragmatic: it is not easy to construct a translation that would admit the inconsistent stable model only when the program does not have any consistent stable models. If we did not want to add a great number of atoms and rules in the program, we would likely have to use some completely altered proof system.

In most practical applications there is no appreciable difference between a program with no stable models and a program with an inconsistent stable model—we do not have a valid solution for our problem in either case.

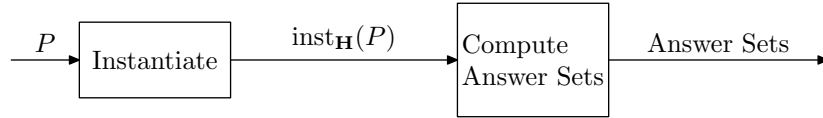


Figure 8.1: Computing answer sets

8 IMPLEMENTATION ISSUES

In this chapter we examine issues relating to implementing ω -restricted CCPs. Some of these subjects have been touched before in Chapters 4 and 7 but here we go into more detail.

We can divide the problem of finding the answer sets of a program into two stages: instantiation and model generation that are illustrated in Figure 8.1. In the first stage we create the relevant instantiation of the problem and in the second stage we compute the answer sets of the instantiation. Most current ASP systems work with this bottom-up principle.

In this work we will concentrate on instantiation. Our goal is to examine how we can start from a full language user program and translate it to a set of ground rules that can be given as an input to some inference engine.

We will continue by first examining the roles of the full and basic languages in more detail and how they correspond to the front- and back-ends of compiler technology. Then, we continue with an overview of the implementation architecture. In Sections 8.4 and 8.3 we present algorithms for computing the domain model and the relevant instantiation of an ω -restricted program. We then conclude this section by examining how we can generate rules that can be used with *smodels*.

8.1 A HIERARCHY OF LANGUAGES

When we are implementing ω -restricted cardinality constraint programs, we work with several different languages. The idea is that we have a hierarchy of languages (see Figure 8.2) with different expressive powers. At each level we use a transformation that rewrites the program in terms of the next simpler language.

The user writes his or her program using the full language. The system translates it into a basic program that is then instantiated. In the last phase we give the instantiated program to some answer set solver.

This hierarchical approach makes it possible to have a relatively simple implementation for the full language since every individual transformation is simple. Also, it allows us to substitute different transformations and tools in the process if we need to extend the semantics or want to use different solvers to do the actual answer set computation. We also can add rule rewriting to different phases of program processing. In rule

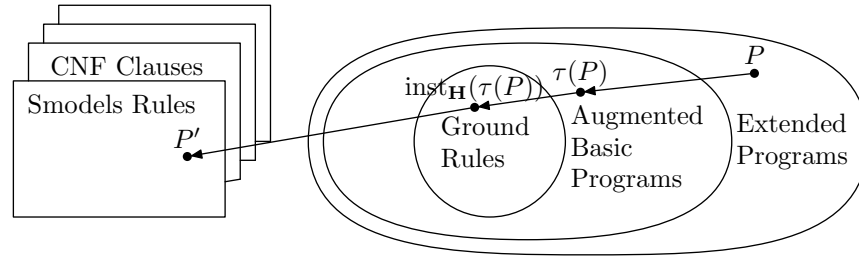


Figure 8.2: The language hierarchy

rewriting we substitute one or more rules of the program with rules that are logically equivalent but that can be processed more efficiently.

Conceptually we can liken the languages to different phases of a compiler [1]. A compiler is divided into two parts, a *front end* that reads in the user program and transforms it into an internal representation. Then, a *back end* takes the internal representation and creates the object code from it. With many compilers it is possible to combine different front ends with different back ends. For example, the free gcc-compiler¹ has front ends for compiling C, Fortran, Java, and several other programming languages, and back ends for generating code for almost all computer architectures in current use.

We use basic programs that are augmented with integral ranges as our internal representation. We call these *augmented basic programs* (ACCP). We treat the integral ranges as a special case because they are semantically complex enough that they are cumbersome to add to the basic language but that cannot be translated until we know what values our numerically interpreted constants take.

Our front end takes in the user program and translates it into the internal representation. The back end then instantiates it and translates it into a form accepted by the inference engine.

In this work we translate the instantiated rules into a form that can be used with *smodels* [174]. This translation is simple since *smodels* rules are essentially a subset of our ground basic rules. If we want to use some other solver, we can do it by using a different back end. For example, we could translate ground programs into propositional clauses and use a SAT solver to compute the answer sets [101].

8.2 OVERVIEW OF THE IMPLEMENTATION ARCHITECTURE

We will now examine the two phases of program instantiation. The general architecture for the instantiator is shown in Figure 8.3. The front end reads in the program and transforms it into a basic program and the back end instantiates it into a set of ground rules.

¹Gnu Compiler Collection, available at <http://gcc.gnu.org>.

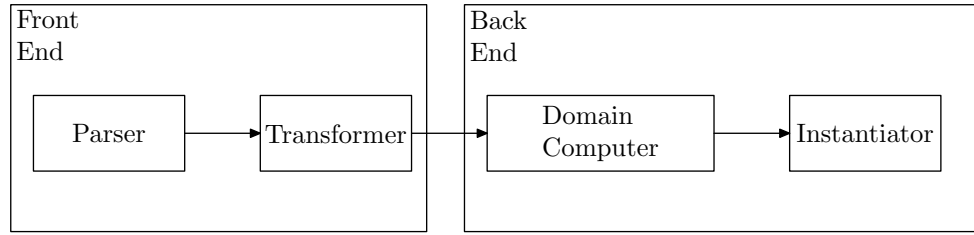


Figure 8.3: General level architecture

```

function instantiate(Program  $P$ )
     $D := \text{create-domain-model}(P)$ 
     $P_G := \text{instantiate-nondomain}(P, D)$ 
return  $P_G \cup F(D)$ 
endfunction

function instantiate-nondomain(Program  $P$ , Domain Model  $D$ )
     $P_G = \emptyset$ 
    foreach  $p \in \mathcal{P}(P)$  do
         $P_G := P_G \cup \text{instantiate-relevant}(p, D)$ 
    endfor
return  $P_G$ 
endfunction

```

Figure 8.4: The instantiation algorithm

8.2.1 Front End

Parser

The *parser* reads in the user program and constructs a tree-representation of the full language program. At the same time it creates a number of symbol tables where every predicate and function symbol is stored with their arities as well as every constant and variable. The parser is also responsible for catching syntax errors that occur in the program.

Transformer

The *transformer* takes the tree-representation generated by the parser and applies the transformations from Chapter 7 to create the corresponding basic program. When a transformation creates new rules, they are kept together so that we can later instantiate them at the same time.

8.2.2 Back End

Figure 8.4 shows a simple pseudo-code algorithm for the back end of the implementation. The algorithm first computes the domain model of the program, and then instantiates all rules for non-domain predicates. Finally, it returns the instantiated rules as well as the domain model expressed as facts.



Figure 8.5: Overview of rule instantiation

Domain Computer

The first part of the back end is the *domain computer*. It first identifies the domain predicates of the program and then computes the domain model. The domain computer also checks that all rules in the program are ω -restricted.

In the current implementation the domain computer uses the instantiator during domain computation to create the relevant instantiations of the domain predicates.

Instantiator

The *instantiator* goes through all rules of non-domain predicates and instantiates them. The global variables are instantiated first, and then the local variables are expanded.

8.3 INSTANTIATING RULES

We start our detailed discussion by examining what happens when we instantiate a rule. In this section we assume that we have already computed the domain model somehow.²

When we instantiate the rules, the general approach is that we first instantiate all global variables of the rule and only after that expand the local variables. This process is illustrated in Figure 8.5.

In Figure 8.6 we see an algorithm for creating the relevant instantiation of a predicate symbol. This algorithm works in one pass and it visits each rule only once and creates its full instantiation at the same time.

We examine how the algorithm works in detail in Section 8.3.3. Before we can do it, we have to describe auxiliary functions for handling domain predicates and variable bindings.

8.3.1 Variable Bindings

Each global variable that occurs in a rule has some domain literal that sets its value. When a variable occurs in more than one domain literal, we choose one of them to be the setter. Which choice we take depends on the nature of the extensions of the domain literals. We will visit this question again later in this section.

We will examine the domain literals $body_{\mathcal{D}}(R)$ in a fixed order where each literal binds its variables that have not been bound by previous

²The details for computing the domain model are given in Section 8.4.

```

function instantiate-relevant(Predicate Symbol  $p$ , Domain Model  $D$ )
  Let  $P = \emptyset$ 
  foreach rule  $R$  of  $p$  do
    if  $R$  has global variables then
       $P := P \cup \textit{instantiate-rule}(R, D)$ 
    else
       $P := P \cup \{\textit{instantiate-local}(R, D)\}$ 
    endif
  endfor
  return  $P$ 
endfunction

function instantiate-rule(Rule  $R$ , Domain Model  $D$ )
  Let  $L$  be an array of literals
  Let  $P = \emptyset$ 
   $L := \textit{choose-order}(\textit{body}_D(R))$ 
   $k := 1$ 
   $\sigma := \emptyset$ 
  while  $k > 0$  do
     $I := \textit{get-next-instance}(L[k], \sigma, D)$ 
    if  $I = \text{null}$  then
       $\textit{reset-instances}(L[k])$ 
       $k := k - 1$ 
      if  $k > 0$  then
         $\sigma := \textit{remove-binding}(L[k], \sigma)$ 
      endif
    elseif  $\textit{can-bind}(L[k], I, \sigma)$ 
       $\sigma := \textit{bind-variables}(L[k], I, \sigma)$ 
      if  $k = |L|$  then
         $P := P \cup \{\textit{instantiate-local}(R, \sigma)\}$ 
         $\sigma := \textit{remove-binding}(L[k], \sigma)$ 
      else
         $k := k + 1$ 
      endif
    endif
  endwhile
  return  $P$ 
endfunction

```

Figure 8.6: Instantiating rules

literals. We use the array notation to denote this order where $L[1]$ is the first domain literal L_1 and $L[|body_D(R)|]$ the last one.

When an iterator returns a new ground instance, we use it to bind the variables that the literal sets. Before we call the iterator the next time we have to clear the existing bindings. We use three different functions for this process:

- *can-bind*(L, I, σ): checks whether the new ground instance I of the domain predicate L is compatible with the existing variable substitution σ ;
- *bind-variables*(L, I, σ): returns a new substitution that extends σ by binding the variables of L into the corresponding terms of the ground instance I ; and
- *remove-binding*(L, σ): removes those bindings from σ that were introduced by the domain literal L and leaves bindings that existed before intact.

These functions are straightforward to implement and we can simply think of the bindings as a set of variable-value pairs so we do not go deeply into their details.

The current implementation defines a substitution as an array where we have one cell for every variable. The contents of a variable binding are then stored in the cell and unbound variables are denoted by a special null value.

8.3.2 Interface to the Domain Model

The functions *instantiate-rule* and *instantiate-local* need an interface to the domain model. We do it by defining *iterators* that go through the extensions of the domain predicates. The iterators have to have two operators defined for them:

1. *get-next-instance* that returns the next ground instance of a domain predicate; and
2. *reset-instances* that returns an iterator back to the start.

As long as we have these two functions available, we can choose the actual implementation quite freely. We will examine briefly three possible approaches.

Get Next Instance

The function *get-next-instance*(L, σ, D) returns the next ground instance of the domain literal L , or **null** if there are no more instances. The function takes also the existing variable bindings σ as input so that it can use them to guide the iteration.

The simplest possible iterator, *get-next-instance₀* does not use σ at all but instead always returns the complete extension one instance at a time.

Another possibility is that we choose one of the variables that occurs in a domain literal to be an index value [202]. We examine the substitution σ

Iteration	$get\text{-}next\text{-}instance_0$	$get\text{-}next\text{-}instance_1$	$get\text{-}next\text{-}instance_n$
1	$d_3(1, 1, 1)$	$d_3(1, 1, 1)$	$d_3(1, 2, 1)$
2	$d_3(1, 2, 1)$	$d_3(1, 2, 1)$	$d_3(1, 2, 2)$
3	$d_3(1, 2, 2)$	$d_3(1, 2, 2)$	null
4	$d_3(2, 1, 1)$	null	
5	null		

Table 8.1: Example behavior of the different iterators

to see what value the index has, and then return only those instances where the value is the same as the existing binding. We call this approach $get\text{-}next\text{-}instance_1$.

The third possibility is that the iterator examines all existing variable bindings and returns those instances where they are all satisfied. We call it $get\text{-}next\text{-}instance_n$.

Example 8.3.1 Suppose that we are instantiating the rule:

$$p(X, Y) \leftarrow d_1(X), d_2(X, Y), d_3(X, Y, Z)$$

with the existing variable bindings $X/1$ and $Y/2$, and we want to iterate through the extension of d_3 that is:

$$\text{ext}(d_3) = \{\langle 1, 1, 1 \rangle, \langle 1, 2, 1 \rangle, \langle 1, 2, 2 \rangle, \langle 2, 1, 1 \rangle\} .$$

The values that the three different iterators return are shown in Table 8.1. It is assumed that $get\text{-}next\text{-}instance_1$ uses the variable X as its index value.

There is a trade off in using the different approaches. A simple iterator finds the next ground instance quickly but it also generates a number of spurious instances that are not compatible with the existing variable bindings. A complex iterator takes a longer time to evaluate but it produces less or even none incompatible instances.

The current implementation uses all three of these basic iterators, though, only a limited form of $get\text{-}next\text{-}instance_n$ is used. We divide the domain literals into three classes:

1. literals with no existing variable bindings;
2. literals that are partially instantiated; and
3. literals that are already fully instantiated.

For the domain literals in the first class we use $get\text{-}next\text{-}instance_0$ and for the second class we use $get\text{-}next\text{-}instance_1$. The domain literals that belong to the third class do not create new instances but instead prune out existing variable bindings. When we have bound all variables of such a literal, we can just check whether that instance occurs in its extension.

When we use $get\text{-}next\text{-}instance_1$, we have to choose which argument position to use for the index. Creating an index for a relation is a computationally heavy operation [202] and we do not want to create more

indices than necessary. When we examine a domain literal, we try to pick a position that has a preexisting index. If there is no such a position, we choose the first variable that has an existing binding and create an index for it. As a special case we prefer those argument positions with constants instead of variables.

Example 8.3.2 Suppose that a program P has the following rule:

$$\{p(X, Y)\} \leftarrow d_1(X), d_2(Y, X, Z), d_3(X, Y, Z)$$

and that we use the ordering where the domain literals are examined in the order that they occur in the rule body.

For this rule we choose $\text{get-next-instance}_0$ for d_1 since d_1 sets the value for X . The next literal d_2 is partially instantiated so we use $\text{get-next-instance}_1$. Now we do not have a choice on the index value as X is the only bound variable. Thus, we create an index for the second argument position of $d_2/3$.

This literal binds both of the remaining variables, so the only thing that is left for d_3 is to check whether the ground instance $d_3(X, Y, Z)\sigma$ is true or not.

Suppose that P has also the rule:

$$\{r(X, Y, Z)\} \leftarrow d_1(X), d_4(Y), d_2(X, Y, Z) .$$

Here we use $\text{get-next-instance}_0$ for both d_1 and d_4 , and $\text{get-next-instance}_1$ for d_2 . Now we can choose to use either X or Y as the index variable. Since we already used the second argument position as the index for the previous rule, we use it again and take Y as the index variable.

8.3.3 Instantiate Relevant

The function *instantiate-relevant* combines together the ground instances of every rule for a given predicate symbol p . If a rule R does not have any global variables at all, we just instantiate its local variables and add the result to the instantiation. If it has global variables, we call *instantiate-rule*(R) to create the instantiation.

The function returns the set of ground instances of rules for p as its result. In practice, we often do not want to keep the whole set in memory at the same time, unless we want to do some additional rule rewriting later. Instead, we can output the rule as soon as it is instantiated and then forget it. This saves memory during the execution as we do not need to store the complete instantiation at the same time.

8.3.4 Instantiate Rule

The function *instantiate-rule* takes as its input one rule and it then instantiates the global variables that occur in it before passing the partial substitution on to other functions that handle the local variables.

In the first part of the function we put the domain literals in some order and store them in the array L , and the main loop goes through L

k	I	σ	Action
1	$d_1(1)$	\emptyset	Bind $X/1$, advance k
2	$d_2(1, 1)$	$\{X/1\}$	Bind $Y/1$, advance k
3	$d_3(1, 2, 1)$	$\{X/1, Y/1\}$	Incompatible Y
3	null	$\{X/1, Y/1\}$	Backtrack k , clear Y
2	$d_2(1, 2)$	$\{X/1\}$	Bind $Y/2$, advance k
3	$d_3(1, 2, 1)$	$\{X/1, Y/2\}$	Bind $Z/1$, call <i>instantiate-local</i> for $p(1, 2) \leftarrow d_1(1), d_2(1, 2), d_3(1, 2, 1)$, clear Z .
3	null	$\{X/1, Y/2\}$	Backtrack k , clear Y
2	$d_2(2, 1)$	$\{X/1\}$	Incompatible X
2	null	$\{X/1, Y/2\}$	Backtrack k , clear X
1	null	\emptyset	Algorithm complete

Table 8.2: Example of the *instantiate-rule* Algorithm

one literal at a time. The variable k is an index to the array that keeps track on which literal we are currently examining.

In the beginning of the loop we generate one ground instance of $L[k]$ and store it into the variable I . If such a value exists, we first check if it is compatible with our existing variable binding. If so, we bind the free variables and advance to the domain literal $k + 1$ unless we are already at the value $k = |L|$ and have values for all global variables. Then we call *instantiate-local* to handle the local variables. We then add the resulting rule to the instantiation, clear the latest variable bindings, and return to the beginning of the loop without altering k .

If the instance I is not compatible with existing bindings, we go back to the beginning of the loop and try again.

If $I = \mathbf{null}$, we have gone through the complete extensions of $L[k]$ and have to backtrack. We reset the iterator of $L[k]$ and remove the variable bindings introduced by $L[k - 1]$. The reason why we remove the bindings of the previous literal is that at the k th level we are trying to extend the bindings that were generated in the first $k - 1$ th levels. If that fails, we have to remove the latest binding so that we can set new values for those variables and then try again at the k th level.

Example 8.3.3 Consider again the rule:

$$p(X, Y) \leftarrow d_1(X), d_2(X, Y), d_3(X, Y, Z)$$

Suppose that this time the extensions of the domain literals are:

$$\begin{aligned} \text{ext}(d_1) &= \{d_1(1)\} \\ \text{ext}(d_2) &= \{d_2(1, 1), d_2(1, 2), d_2(2, 1)\} \\ \text{ext}(d_3) &= \{d_3(1, 2, 1)\} \end{aligned}$$

and we again go through them in the numerical order.³ Table 8.2 shows the steps that *instantiate-rule* takes while working on this rule when we use the simplest iterator.

³In this case it would be most efficient to handle d_3 first since it has only one instance and it gives values for all variables at the same time.

8.3.5 Ordering Domain Literals

The algorithm *instantiate-rule* in Figure 8.6 computes the join of the extensions of domain literals in the rule body one tuple at a time.⁴ The order that we choose for the literals is very important for practical performance since the number of computation steps we take in the main loop is dependent on the size of the intermediate products of the join. Here it is possible to use standard database algorithms for deciding a good join ordering [202, 110, 68]. We do not go into the details of such algorithms and only give an example that illustrates why the order matters and give a couple of simple heuristics for deciding it.

Example 8.3.4 *Suppose that we want to instantiate the rule:*

$$p(X, Z) \leftarrow d_1(X), d_2(Y), d_3(X, Y, Z)$$

where the extensions of the domain predicates are:

$$\begin{aligned} \text{ext}(d_1) &= \{d_1(x) \mid x \in [1 \dots 1000]\} \\ \text{ext}(d_2) &= \{d_2(y) \mid y \in [1 \dots 1000]\} \\ \text{ext}(d_3) &= \{d_3(x, y, z) \mid x, y, z \in [1 \dots 10]\} \end{aligned}$$

This rule has $10 \cdot 10 \cdot 10 = 1000$ relevant instances. With the ordering $\langle d_1, d_2, d_3 \rangle$ we essentially first create the Cartesian product $\mathbf{R}_{d_1} \times \mathbf{R}_{d_2}$ whose size is $|\mathbf{R}_{d_1}| \cdot |\mathbf{R}_{d_2}| = 1,000,000$ even though only 100 tuples from it are compatible with the extension of d_3 .

However, if we choose to handle d_3 first we do not have to examine any superfluous tuples at all and we may handle d_1 and d_2 in either order.

The current implementation uses the following heuristics to determine the order:

1. Sort the domain literals in order $\langle A_1, A_2, \dots, A_n \rangle$ in such a way that $\text{ext}(\text{pred}(A_i)) \leq \text{ext}(\text{pred}(A_{i+1}))$ for all $1 \leq i < n$.
2. If the extensions of A_i and A_{i+1} are as large and A_{i+1} binds more variables than A_i , swap their places.
3. If all variables of a literal A_i are bound by earlier literals $\langle A_1, \dots, A_k \rangle$ ($k < i$), then move A_i into the $(k+1)$ th position and shift the other literals by one to make space for it.

These heuristics do not guarantee that the join order is optimal and it is possible to construct an example where we have to do a lot of unnecessary work compared with an optimal order.

Example 8.3.5 *Consider the rule:*

$$h(V, X, Y, Z) \leftarrow d_1(X, V), d_2(X, Y, Z), d_3(Y)$$

⁴This is proved in Section 8.3.9.

```

function instantiate-local(Rule  $R$ , Substitution  $\sigma$ , Domain Model  $D$ )
  Let  $S$  be the set of conditional literals in  $R$ 
   $E := \emptyset$ 
  foreach  $L : A \in S$  do
     $E := E \cup \text{expand-condition}(L : A, \sigma, D)$ 
  endforeach
   $R' := \text{replace-conditionals}(R\sigma, S, E)$ 
  return  $R'$ 
endfunction

function expand-condition(Conditional Literal  $L : A$ , Substitution  $\sigma$ ,
                           Domain Model  $D$ )
   $E := \emptyset$ 
   $I := \text{get-next-instance}(A, \sigma, D)$ 
  while  $I \neq \text{null}$  do
    if can-bind( $A, I, \sigma$ ) then
       $\sigma' := \text{bind-variables}(A, I, \sigma)$ 
       $E := E \cup \{L\sigma' : A\sigma'\}$ 
    endif
     $I := \text{get-next-instance}(A, \sigma)$ 
  endwhile
  return  $E$ 
endif

```

Figure 8.7: Instantiating local variables

where the sizes of the extensions of the domain literals d_i are:

$$\begin{aligned}
 |\text{ext}(d_1)| &= |\text{ext}(d_2)| = 1000 \\
 |\text{ext}(d_3)| &= 10000
 \end{aligned}$$

The extensions of d_1 and d_2 are as large so the heuristics chooses d_2 first since it binds three variables while d_1 binds only two. Even though d_3 has the largest extension, it is moved to the second position since all its lone variable has already been bound while d_1 still has a free variable. The complete order is then:

$$\langle d_2, d_3, d_1 \rangle .$$

8.3.6 Instantiate Local

The function *instantiate-local* goes through every conditional literal in the rule R , expands it and replaces the conditional literals with their expansions.

The literal expansion works in an analogous way to *instantiate-rule*, except that in this case we have only one literal to consider.⁵ We iterate

⁵Note that we can implement the extension of having multiple conditions in a conditional literal (Section 7.3.4) also by changing *expand-literal* to go through all of them in a similar way how *instantiate-rule* handles multiple domain literals.

through the extension of the condition and add the compatible ground instances of the main literal to the expansion.

The auxiliary function *replace-conditionals*(R, S, E) replaces all conditional literals in the set S by their expansions E .

Example 8.3.6 Consider the rule:

$$\leftarrow 1 \{Y.p(X, Y) : d_2(X, Y)\}, d_1(X) .$$

where the extensions of the domain predicates are as in Example 8.3.3.

Calling *instantiate-rule* we get the global variable binding $X/1$. When we go over the extension of d_2 with *get-next-instance*₁, we find that the two compatible instances are $d_2(1, 1)$ and $d_2(1, 2)$, so we get the set:

$$E = \{p(1, 1) : d_2(1, 1), p(1, 2) : d_2(1, 2)\}$$

This gives us the ground instance:

$$\leftarrow 1 \{p(1, 1) : d_2(1, 1), p(1, 2) : d_2(1, 2)\}, d_1(1) .$$

In practice, we already know that both conditions are true so we could leave them out and instead use the ground instance:

$$\leftarrow 1 \{p(1, 1), p(1, 2)\}, d_1(1) .$$

8.3.7 Handling Function Symbols

We left out the handling of function symbols from the pseudocode for *instantiate-rule* in Figure 8.6. In this section we examine how we can implement them.

Interpretation Functions

We write an interpretation function *eval_f* for every function symbol f in our language. The implementation contains such functions for all built-in function symbols of the standard interpretation 3.9 and there is a simple application programming interface that allows a user to define his or her own functions.

Compound Terms

We introduce a new variable X_t for every compound term t that occurs in a rule. Then, we replace every occurrence of t with X_t and add the built-in predicate $X_t = t$ to the rule body.

Example 8.3.7 Consider the rule:

$$p_1(X, X + Y) \leftarrow d_1(X), d_2(Y), p_2(X + Y) ,$$

where d_1 and d_2 are the domain literals. We translate this into:

$$p_1(X, X_{X+Y}) \leftarrow d_1(X), d_2(Y), X_{X+Y} = X + Y, p_2(X_{X+Y}) .$$

The main reason why we use this two-step approach is that it makes it easy to apply substitutions to literals. As the substitutions are implemented as an array S of constant terms, we can just look at the contents of $S[i]$ when we want to substitute the variable X_i .

Herbrand Interpretations

If a function symbol f/n has the Herbrand interpretation, its interpretation function creates a new ground term $f(t_1, \dots, t_n)$ based on the existing variable bindings.

When we store $f(t_1, \dots, t_n)$ in the array S of variable bindings, we transform the n -ary compound term into the 0-ary constant $\underline{f(t_1, \dots, t_n)}$. This is again done to simplify applying the substitutions.

Built-In Predicates

The standard interpretation defines built-in predicates for the usual relational operators ($=$, \neq , $<$, etc.). We define an interpretation functions $eval_p$ for each built-in predicate symbol analogously to the case of built-in functions. A function call $eval_p(t_1, t_2)$ returns **true** if $p(t_1, t_2)$ is true, and **false** otherwise.

During instantiation we use built-in predicates as a test and discard all ground instances for which $eval_p(t_1, t_2)$ returns **false**.

We want to evaluate these predicates as soon as possible. This means that if the last variable in $p(t_1, t_2)$ is bound on the k th iteration of *instantiate-rule*, then we call $eval_p$ before we proceed to the $(k + 1)$ th iteration.

Example 8.3.8 Consider the rule:

$$p(X, Y, Z) \leftarrow d_1(X), d_2(Y), d_3(Z), X > 2, X < Y + 1$$

with the domain literal ordering $\langle d_1, d_2, d_3 \rangle$. In this rule we can evaluate $X > 2$ immediately after we bind X and $X < Y + 1$ when we have bound both X and Y .

This means that we do not have to find a value for Y unless X is greater than 2. Similarly, we do not have to iterate over d_3 if $X > Y$.

8.3.8 Improving the Algorithm

The *instantiate-rule* algorithm is simple and easy to implement and it allows us to handle all rules in the program in an uniform way. However, there are situations where its performance is less than optimal. In this section we identify several of these cases and discuss how we could improve the behavior.

Superfluous Ground Instances

The *instantiate-rule* algorithm may create more ground instances than necessary. It may create rules that are essentially duplicates of each other as well as rules whose bodies cannot be satisfied in any stable model of the program.

For example, consider the rule:

$$p(X) \leftarrow d(X, Y) .$$

Here $p(X)$ will be true if there is at least one Y for which $d(X, Y)$ is true but the algorithm will instantiate this rule for every distinct value

of Y . We could handle these by adopting the standard techniques for computing projections of joins [203].

The second possibility is that when we have a recursive domain predicate, we may create a large number of rules whose bodies cannot be satisfied. Suppose that we have a domain predicate p that is defined with the rules:

$$\begin{aligned} d(0..1000) &\leftarrow \\ p(0) &\leftarrow \\ p(X+1) &\leftarrow p(X), d(X), X > 1 \end{aligned}$$

The body of the second rule is not satisfied for any value of X , but we still create a ground instance for all of them. Unfortunately, we cannot do much for this weakness unless we want to make drastic changes in the algorithm.

The function *instantiate-rule* looks at only one rule at a time. The advantage of this feature is that we need to examine each rule just once during instantiation. The cost that we have to pay is that we cannot detect these kinds of dependencies between rules.

Rule Rewriting

In some problem domains we have many rules that have a similar or even the same set of domain literals. If we examine each such a rule separately, we have to create the same natural join many times. For example, if we have the rules:

$$\begin{aligned} p(X, Y, Z) &\leftarrow \text{not } q(X, Y, Z), d_1(X, Y), d_2(Y, Z) \\ q(X, Y, Z) &\leftarrow \text{not } p(X, Y, Z), d_1(X, Y), d_2(Y, Z) \end{aligned}$$

we have to create the join $d_1 \bowtie d_2$ separately for both rules.

In *rule rewriting* we replace a rule with another that is computationally more efficient. In the above example we might define a new domain predicate:

$$d_3(X, Y, Z) \leftarrow d_1(X, Y), d_2(Y, Z)$$

and rewrite the two rules into:

$$\begin{aligned} p(X, Y, Z) &\leftarrow \text{not } q(X, Y, Z), d_3(X, Y, Z) \\ q(X, Y, Z) &\leftarrow \text{not } p(X, Y, Z), d_3(X, Y, Z) \end{aligned}$$

One thing to note is that when we restrict rewriting to the domain literals, we do not get any new ground atoms into the non-domain part of the instantiation. This means that the instantiation-time optimizations do not make the task of finding answer sets more difficult.

There are many standard techniques [202, 203] that can be applied to identify the common parts of the domains of the rules and to evaluate whether the savings that we get outweigh the costs inherent in computing and storing the extension of the new predicate.

In cases where several rules have identical domain literals, we do not need to create and store a new domain predicate explicitly. Instead, we can alter *instantiate-rule* so that we instantiate all of them at the same time. Whenever we find a ground instance, we call *instantiate-local* for every rule before advancing to the next instance.

Backjumping

In the main loop of *instantiate-rule* we always backtrack one level of variable bindings at the time. This may cause a problem if we chose a poor order for the domain literals. If it happens that the first variable binding that we made conflicts with the last binding, we have to create the complete natural join in the middle in vain.

It should be possible to augment the algorithm to support *backjumping* [8, 163] where we remove more than one variable binding at one time. Consider the rule:

$$p(X, Y, Z) \leftarrow d_1(X), d_2(Y), d_3(X, Y, Z)$$

with the domain predicate extensions:

$$\begin{aligned} d_1 &: \{d_1(1), d_1(2)\} \\ d_2 &: \{d_2(1), d_2(2)\} \\ d_3 &: \{d_3(2, 2, 2)\} \end{aligned}$$

Suppose that we first bind X to 1 and then Y to the same value. Then we notice that there are no compatible instances for d_3 . Moreover, it has no instances at all that agree with the value for X so there is no need to consider the second possible value for Y and we can instead remove the bindings for both X and Y at the same time.

8.3.9 Correctness of Rule Instantiation Algorithm

The instantiation algorithm differs from the formal definition of the semantics in that we handle global and local variables in a different order.

The formal definitions of Chapter 3 are ordered so that we first expand the conditional literals and only after that instantiate the global variables. In practice, it is more efficient to do it in the other order: when we know the values of the global variables, we can restrict the number of ground instances of the main literal of the conditional literal and create only those instances that are compatible with the existing bindings.

Next, we argue formally that the function *instantiate-rule*(R, D) produces the relevant instantiation of the rule R . In Section 5.8 we defined a relation $\mathbf{R}_{R,D}$ over the domain literals of a rule R and showed that a tuple $\langle a_1, \dots, a_n \rangle \in \mathbf{R}_{R,D}$ exactly when there is a corresponding ground rule in the relevant instantiation of the rule.

We can think that *instantiate-rule* creates a relation over the extensions of the domain predicates one tuple at a time. We show that this relation is actually the same relation as $\mathbf{R}_{R,D}$. Thus, *instantiate-rule* creates all the substitutions for global variables that satisfy all domain literals.

Then, the function *instantiate-local* computes the evaluation of the conditional literals that occur in the rule. We show that a ground conditional literal $L\sigma : A\sigma$ is in the set E that *expand-condition* computes for $X.L : A$ exactly when $D_\omega \models A\sigma$ and σ agrees with the global variable bindings.

Definition 8.3.1 Let P be an ω -restricted program, $R \in P$ be a rule where $\text{Var}_g(R) = \{X_1, \dots, X_k\}$, $\{p_1, \dots, p_n\}$ be the set of predicate symbols occurring in its domain literals, and Σ be the set of substitutions produced by $\text{instantiate-rule}(R, D_\omega)$. Then, the relation $\mathbf{R}_I[X_1, \dots, X_k]$ is defined as follows:

$\langle a_1, \dots, a_k \rangle \in \mathbf{R}_I[X_1, \dots, X_k]$ if and only if there exists $\sigma \in \Sigma$ such that $\sigma(X_i) = a_i$ for $1 \leq i \leq k$.

Proposition 8.3.1 If P is an ω -restricted program, then $\mathbf{R}_I = \mathbf{R}_{R, \mathcal{D}}$ for all rules $R \in P$.

Proof. Suppose that $\langle a_1, \dots, a_k \rangle \in \mathbf{R}_{R, \mathcal{D}}$. By Proposition 5.8.1 there exists a substitution σ such that $\sigma(X_i) = a_i$ for all $1 \leq i \leq k$ and $D_\omega \models A\sigma$ for all $A \in \text{body}_{\mathcal{D}}(R)$.

Let $\langle A_1, \dots, A_n \rangle$ be an ordering of the domain literals in $\text{body}_{\mathcal{D}}(R)$. Since $D_\omega \models A_1\sigma$, $\text{get-next-instance}(A_1, \emptyset, D_\omega)$ eventually returns a substitution σ_1 such that $\sigma_1 = \sigma|_{\text{Var}(A_1)}$.⁶ Similarly, as $D_\omega \models A_2\sigma$, $\text{get-next-instance}(A_2, \sigma_1, D_\omega)$ eventually returns σ_2 such that $\sigma_2 = \sigma|_{(\text{Var}(A_1) \cup \text{Var}(A_2))}$. Continuing this, we eventually find a substitution $\sigma_n = \sigma$, so $\langle a_1, \dots, a_k \rangle \in \mathbf{R}_I$.

Conversely, if $\langle a_1, \dots, a_k \rangle \in \mathbf{R}_I$, we have found an admissible combination of ground instances for $\langle A_1, \dots, A_n \rangle$. So, we have a substitution σ such that $D_\omega \models A_i$ for all domain literals A_i . Applying Proposition 5.8.1 again we conclude that $\langle a_1, \dots, a_k \rangle \in \mathbf{R}_{R, \mathcal{D}}$. \square

Proposition 8.3.2 Let P be an ω -restricted program, $R \in P$ a rule in it and $X.L : A$ be a conditional literal in R . Then, a ground conditional literal $L' : A'$ belongs to the set E computed by $\text{expand-condition}(L, \sigma, D_\omega)$ if and only if there exists a substitution $\sigma_X \in \text{subs}(X, U_{\mathbf{H}}(P))$ such that $D_\omega \models A\sigma_X\sigma$ and $L' = \sigma_X\sigma$.

Proof. Suppose that a ground conditional literal $L' : A' \in E$. This is possible only if there exists a substitution σ' such that it is the result of the function call $\text{bind-variables}(A, I, \sigma)$ where I is a ground instance of $A\sigma$ in D_ω . Thus, $D_\omega \models A\sigma\sigma'$. Since $\text{Var}(\sigma) \cap \text{Var}(\sigma') = \emptyset$ and since σ and σ' are ground substitutions, we can reorder them to get $D_\omega \models A\sigma'\sigma$. Finally, we restrict σ' to the set of local variables X to get $\sigma_X = \sigma'|_X$. Now $D_\omega \models A\sigma_X\sigma$.

Next, suppose that there is a ground instance $A' \in D_\omega$ where $A' = A\sigma_X\sigma$. As σ_X and σ do not operate on same variables, we can reorder them to get $D_\omega \models A\sigma\sigma_X$. So the test $\text{can-bind}(A, I, \sigma)$ succeeds for the ground instance $I = A\sigma\sigma_X$, so $\text{bind-variables}(A, I, \sigma)$ returns $\sigma' = \sigma\sigma_X$ and we add $L\sigma' : A\sigma'$ into E . \square

8.4 DOMAIN COMPUTATION

We will now examine how we can compute the domain model of a CCP. We already saw an algorithm for identifying the domain predicates and

⁶Where $\sigma|_X$ denotes the restriction of σ to the variables of the set X .

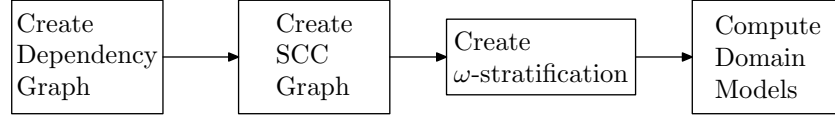


Figure 8.8: Domain computation

```

function create-domain-model(Program  $P$ )
  Graph  $G := \text{create-dependency-graph}(P)$ 
  Graph  $S := \text{create-SCC-graph}(G)$ 
  Domain Model  $D := \text{visit-components}(P, G)$ 
  return  $D$ 
endfunction
  
```

Figure 8.9: Algorithm for Computing the Domain Model

computing the domain model in Section 4.4. Another way to do it would be to use Datalog-style relation evaluation to create the domain model in a way similar to the one presented in Section 5.8 after we have identified the domain program.

In this section we present a third method that uses the *instantiate-relevant* function as a part of the domain computation. We examine one strongly connected component of the dependency graph at a time, instantiate the rules that occur in it, compute its least model, and add it to the domain model.

This approach has the advantage that it allows the back end to handle all rules in a program in a uniform way. We go through the rules one by one instantiating them, and then either compute the least model if the rule happens to belong to the domain program or output the rule if it belongs to the ω -program.

However, this approach has also the disadvantage that when we have recursive domain predicates, we may end up doing unnecessary work instantiating rules whose bodies are never satisfied. This is a weakness that does not occur in algorithms based on database techniques.

The general approach for the domain computation is shown in Figure 8.8. It consists of four phases:

1. creating the dependency graph;
2. creating the SCC graph;
3. creating an ω -stratification; and
4. computing the domain model.

The first two phases are straightforward. We create the dependency graph by going systematically through all rules in the program and adding all positive and negative dependencies that we encounter. Then, we

```

Let visited be an array with  $|V|$  elements
Let is-domain be an array with  $|V|$  elements
Set visited[v] := false for all  $v \in V$ 
Set is-domain[v] := false for all  $v \in V$ 
function visit-components(Program P, SCCGraph  $S = \langle V, E, N \rangle$ )
    Let  $D := \emptyset$  be a domain model
    foreach  $v \in V$  do
        if visited[v] = false then
             $D := D \cup \text{visit}(v, D, S)$ 
        endif
    endfor
    return D
endfunction

function visit(Component v, Domain Model D, SCCGraph  $S = \langle V, E, N \rangle$ )
    visited[v] := true
    domain := true
    foreach  $v'$  such that  $\langle v, v' \rangle \in E$  do
         $D := \text{visit}(v', D, S)$ 
        if is-domain[ $v'$ ] = false then
            domain := false
        endif
    endfor
    if  $v \notin N$  and domain = true
        is-domain[v] := true
         $D := D \cup \text{compute-extensions}(v, D)$ 
    endif
    return D
endfunction

function compute-extensions(Component v, Domain Model D)
     $P := \emptyset$ 
    foreach  $p \in v$  do
         $P := P \cup \text{instantiate-relevant}(p, D)$ 
    endfor
     $M := \text{least-model}(P)$ 
     $D := D \cup M$ 
    return D
endfunction

```

Figure 8.10: Visiting the strongly connected components

create the strongly connected component graph (SCC graph) using the Tarjan’s algorithm [172, pp. 481–483].

In practice, we are not interested in creating a full ω -stratification. When we instantiate the program, we do not need to know exactly which strata any given predicate symbol belongs to. What we want to ensure is that when we instantiate a domain predicate symbol p , we have already created the partial domain models for all predicate symbols that p depends on. Thus, we can combine the last two phases in one. This algorithm is shown in Figure 8.9.

8.4.1 Traversing the SCC Graph

Figure 8.10 shows the algorithm that we use to compute the domain model based on the strongly connected component graph.

We do a depth-first search through the strongly connected component graph of the program where we first process all successors of a component before examining that component itself. This is the same idea that we used in Section 4.6 to create a strict ω -stratification for a program. The difference is that here we compute the partial domain model that corresponds to a component as soon as we have finished working with its successors.

The practical effect that this causes is that our strict ω -stratification is no longer minimal in the sense that it assigns all predicate symbols to as low stratum as possible. Instead, every component that consists of domain predicates becomes its own stratum. By Theorem 4.4.1 every strict ω -stratification is equivalent so this change does not cause any problems.

Global Arrays

Figure 8.10 shows that we have defined two auxiliary arrays as global variables. There is no particular reason why they could not be carried along as parameters for the *visit* function except that it would make the pseudo-code more difficult to read. The two global arrays are:

- *visited* holds status information for every vertex of the SCC graph S . Its role is to ensure that we visit every component only once. We set the values of *visited*[v] to false for every $v \in V$ at the beginning of the algorithm, and then set it true when we first enter the component during the computation.
- *is-domain* is an array of truth values that tells whether the predicate symbols belonging to that component are domain predicates or not. We initialize this value to **false** at the beginning and then set it true if we find out that the component contains domain predicates.

Visit Components

The function *visit-components* is a simple driver function that ensures that we go through every strongly connected component of the SCC graph.

Visit

The function *visit* checks whether a given strongly connected component v contains domain predicate symbols.

The first thing that we do entering the function is to mark that the component v is visited. Next, we define an auxiliary variable *domain* and set it true. The intuition is that if the value of the variable is still true at the end of the computation, we are possibly dealing with domain predicates.

Next, we recursively visit every successor of the component that we are examining. If it turns out that at least one of them is non-domain, we set the variable *domain* to false to signify that this component is also non-domain.

Finally, we check whether the component contains a negative arc. If we do not have such an arc and *domain* is still true, we mark in the *is-domain* array that this component belongs to the domain program and we proceed to compute the extensions of its predicate symbols. If the component has a negative arc, it contains negative recursion so it has to belong to the ω -stratum.

Compute Extensions

The function *compute-extensions* takes as its argument a strongly connected component v and it computes the extensions of the domain predicates belonging in it and stores them into the partial domain model. This function first instantiates all of the predicate symbols using the function *instantiate-relevant* and then calls a function *least-model* to compute the least model.

Least Model

The function *least-model* computes the least models of the stratum programs. We have already seen one way that we could use to do it, namely the modified *naive-datalog* algorithm from Section 5.8 or one of its more advanced derivatives [203].

In the current implementation we do it in a slightly different way. We compute the full relevant instantiation of the stratum program, then use the *expand* algorithm from the *smodels* inference engine [174] to compute the least model of the instantiation.

The *expand* algorithm takes as its input a set of ground *smodels* rules as well as a partial truth assignment, and it computes the deductive closure of the rules. The algorithm was designed by Patrik Simons and it is based on the linear-time Dowling-Gallier algorithm [51] for computing the least model of a positive normal logic program. Since the algorithm is relatively complex, we do not present its details here and they are published in [174, p. 30].

8.4.2 Correctness of Domain Computation

In the previous section we showed that if we have computed the domain model D_ω , then *instantiate-relevant*(p, D_ω) computes the relevant instantiation for the predicate symbol p . However, we use the same function

in *create-domain-model* to create D_ω . At the first glance of it we have here circular reasoning: *instantiate-relevant* works correctly if we have already computed the thing that we want to compute with it.

Notice that *instantiate-relevant* call in *compute-extensions* takes a partial domain model D as its second argument. By the time we examine a domain predicate p that belongs to the k th stratum, we have already computed the partial domain model D_{k-1} , so we know the full extensions of all domain literals that occur in rules for p and so we can create their relevant instantiations. The next proposition examines this issue with a bit more rigor:

Proposition 8.4.1 *Let P be an ω -restricted CCP, \mathcal{S} its strict ω -stratification, and A a ground atom. If $\mathcal{S}(\text{pred}(A)) = k$, then $D_k^{\mathcal{S}} \models A$ if and only if $D_\omega^{\mathcal{S}} \models A$ where D_k is its k th partial domain model.*

Proof. First, by Definition 5.4.1, $D_k^{\mathcal{S}} \subseteq D_\omega^{\mathcal{S}}$ so if $D_k^{\mathcal{S}} \models A$, then $D_\omega^{\mathcal{S}} \models A$. If $D_k^{\mathcal{S}} \not\models A$ but $D_\omega^{\mathcal{S}} \models A$, then that there is some $k' > k$ such that $D_{k'}^{\mathcal{S}} \models A$. Since all rules for A occur on the k th strata, $D_{k'}^{\mathcal{S}}$ does not contain any rule with A in the head. This causes a contradiction since in Theorem 3.6.1 we proved that every atom that is true needs a supporting rule. \square

8.4.3 Other Possible Approaches

Our method for instantiating a program is not the only possible way and there are other possible approaches.

Database Domain Computation

We could handle the domain computation by traditional database techniques [202, 203, 36] and then use *instantiate-rule* only for the non-domain part of the program.

In essence, we could interpret the domain program as a Datalog program that has been extended with stratified negation and positive cardinality atoms and compute the domain model with deductive database algorithms. We saw a naive algorithm for doing this for ordinary Datalog programs in Section 5.8.

The Discarded Negations Method

It is possible to discard *instantiate-rule* completely and compute the complete instantiation with deductive database methods [161, 68]. The problem that we have to overcome is that the predicates that are defined in terms of negative recursion do not have unique relations that we could attach to them.

The simplest solution is that we temporarily drop all negative literals that take part in a negative cycle in the dependency graph of the program. The resulting program is stratified and we can use standard algorithms to compute its least model. This least model contains all those atoms that we could plausibly derive using the rules of the original program.

Then, we return the negative literals back in place and create those ground instances of the rules where every literal in the rule body is true in the approximated model.

Example 8.4.1 Consider the program:

$$\begin{aligned} d(1) &\leftarrow \\ d(2) &\leftarrow \\ e(3) &\leftarrow \\ p(X) &\leftarrow d(X), \text{not } q(X) \\ q(X) &\leftarrow d(X), \text{not } p(X) . \end{aligned}$$

Using the discarded negations method we would first transform the two non-ground rules into:

$$\begin{aligned} p(X) &\leftarrow d(X) \\ q(X) &\leftarrow d(X) \end{aligned}$$

This gives us the approximated model:

$$M = \{d(1), d(2), e(3), p(1), p(2), q(1), q(2)\} .$$

We can now return the negative literals to the rule bodies and generate the ground instances for $p(1)$, $p(2)$, $q(1)$, and $q(2)$. Since neither $p(3)$ nor $q(3)$ are in M , we do not need to add the corresponding ground rules to the instantiation.

In practice, we do not need to explicitly separate the instantiation and computation of the approximated model. Instead, we can create a ground instance of a rule at the same time as we add its head to the approximated model.

This approach often computes smaller ground instantiation than *instantiate-rule*, since it considers also the extensions of non-domain predicates in the pruning phase when it decides what instances to include and what leave out. The price that we pay for the more compact instantiation is that the algorithm is more complex to implement and it does not create the instantiation in one pass over the rules.

It should be noted that this algorithm does not guarantee a minimal instantiation. It is possible to construct a program where we create a large number of spurious rules. If a rule has a literal in its body that is false in every stable model of the program, it is trivially satisfied and it can never justify any atoms into the stable models so it can be left out. The question whether some literal is true in any stable model of a ground program is **NP**-complete so the problem of deciding if a rule is needed or not is at least that difficult.

Domain Literals and Constraint Satisfaction

In *instantiate-rule* we find the relevant substitutions using the relational model for domain literals. We could interpret the computation also as a constraint satisfaction problem.

A constraint satisfaction problem (CSP) consists of a set of *variables* $X = \{X_1, \dots, X_n\}$, a set of *values* $V(X_i) = \{a_1, \dots, a_n\}$ for every variable X_i , and a set of *constraints* $C = \{C_1, \dots, C_k\}$ where each $C_i \subseteq X_1 \times \dots \times X_n$ is a relation that defines acceptable value combinations. The goal is to find the set of tuples $\langle a_1, \dots, a_n \rangle$ such that $a_i \in V(X_i)$ for all $i \in [1, n]$ and $\langle a_1, \dots, a_n \rangle \in C_i$ for all $C_i \in C$.

When we express the instantiation problem in CSP terms we take the set of global variables of a rule as our set of variables. The constraints are the relations \mathbf{R}_p of domain literals that occur in the body.⁷ We have two choices on how we define the sets of possible values for variables. We can either take the Herbrand universe of the program as the starting point for each variable X_i , or we can compute the intersection of the projections $\pi_{X_i}(\mathbf{R}_p)$. The latter approach is likely to be more efficient in practice and it also allows us to handle function symbols with the Herbrand interpretation.

In practice, there is no great theoretical divide between using the database and CSP approaches. The answer for a constraint satisfaction problem is a relation and in this case the relation will be the same $\mathbf{R}_{R,D}$ that we create with database algorithms. In fact, we could interpret *instantiate-relevant* as a CSP algorithm that does not create explicit representations for the sets of values.

Lambda-Restricted Programs

A program is λ -restricted [81] if it is possible to create a level-mapping $\lambda : \text{Preds}(P) \rightarrow \mathbb{N}$ of its predicate symbols such that every variable that occurs in a rule occurs in a positive literal that belongs in a strictly lower level than the head.

For example, if we have a program:

$$\begin{aligned} a(1) &\leftarrow \\ b(X) &\leftarrow a(X), c(X) \\ c(X) &\leftarrow b(X) \end{aligned} ,$$

there is a level mapping $\lambda(a) = 0$, $\lambda(b) = 1$, and $\lambda(c) = 2$ that satisfies the condition. This program is not ω -restricted since $b/1$ and $c/1$ depend on each other so $\mathcal{S}(c) = \mathcal{S}(b)$ and the last rule does not have a domain literal. Even though $b/1$ and $c/1$ depend on each other, the instances of both of them are restricted by the instances of $a/1$.

The λ -restriction is a generalization of ω -restriction since every strict ω -stratification is a valid λ -mapping. It is possible to create an ω -restricted program that is equivalent to a given λ -restricted one by introducing suitable domain literals to the bodies of rules that are not ω -restricted. In the above example we could add $a(X)$ to the body of the last rule to restrict it. However, if the non ω -restricted rules introduce new terms to the program, then we have to introduce new domain predicates to catch the new terms. For example, if we have:

$$\begin{aligned} b(f(X)) &\leftarrow a(X), \text{not } c(X) \\ c(X) &\leftarrow b(X) \end{aligned} ,$$

⁷In practical situations we do not usually define the constraints to range over the Cartesian product of all variables. Instead, we use only some subset of the variables. when we test whether a tuple satisfies such a constraint we project the relevant values from the tuple and do the check with only them.

we have to define a new domain predicate d to get:

$$\begin{aligned} d(f(X)) &\leftarrow a(X) \\ b(f(X)) &\leftarrow a(X), \text{not } c(X) \\ c(X) &\leftarrow d(X), b(X) . \end{aligned}$$

The GrinGo instantiator [81] implements a back-jumping algorithm for grounding λ -restricted programs. It augments the back-jumping with *binder splitting*. A *binder* is the λ -restriction equivalent of a domain literal; a positive atom that gives the bindings for a variable.

The idea of binder splitting is to reduce the number of rules by removing irrelevant ground instances. For example, if we have the rule:

$$a(X) \leftarrow d(X, Y), \text{not } b(X) ,$$

we will get a ground instance for every different value of Y for which $d(X, Y)$ is true. In binder splitting we replace $d(X, Y)$ by a pair of literals $d(X, -)$, $d(X, Y)$ where $d(X, -)$ represents the projection of $d/2$ to its first argument. When instantiating

$$a(X) \leftarrow d(X, -), d(X, Y), \text{not } b(X) ,$$

we can jump back to $d(X, -)$ immediately after finding one instance for $d(X, Y)$ and we do not have to examine its other instances.

We can get the same effect in ω -restricted programs by defining a new predicate symbol $d'/1$ for the projection. $d/2$ and by replacing $d(X, Y)$ with it:

$$\begin{aligned} a(X) &\leftarrow d'(X), \text{not } b(X) \\ d'(X) &\leftarrow d(X, Y) . \end{aligned}$$

8.5 THE *smodels* RULES

The rules that the instantiator provides are ground CCP rules. When we want to use the *smodels* engine to compute the stable models of CCP programs, we have to translate them into a form that *smodels* understands.

8.5.1 Rule Types

The *smodels* inference engine uses four different kinds of ground rules [174]. The rule types are simplified forms of cardinality constraint rules where there may be at most one cardinality or weight atom in each rule.

Basic Rules

A *smodels basic rule* is a ground normal logic program rule:

$$h \leftarrow a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m$$

where h , a_i , and b_i are all atoms. Constraints with empty heads are handled by substituting the atom \perp to the head and then rejecting every model candidate where \perp is true.

Choice Rules

A *smodels* choice rule is a ground rule:

$$\{h_1, \dots, h_k\} \leftarrow a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m$$

where h_i , a_i , and b_i are all atoms. Note that an *smodels* choice rule cannot have any cardinality atoms in its body.

Constraint Rule

A *smodels* constraint rule is a ground rule:

$$h \leftarrow \text{Card}(1, \{a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m\})$$

where h , a_i , and b_i are all atoms. Each constraint rule has to have a single atom as the head and exactly one constraint atom in the body.

Weight Rule

A *smodels* weight rule is a ground rule:

$$h \leftarrow \text{Weight}(1, \{a_1, \dots, a_n, \text{not } b_1, \dots, \text{not } b_m\})$$

where h , a_i , and b_i are all atoms. As was the case of constraint rules, weight rules need to have exactly one weight atom in the body.

8.5.2 Rule Translation

There are three possible reasons why a ground rule R produced by *instantiate-rule* might not be an *smodels* rule:

1. R may be a choice rule that has cardinality literals in the rule body;
2. R may be a basic rule containing more than one cardinality atom in the body; or
3. R may contain negative cardinality literals in the body.

We can handle all three cases with one transformation. Let R be a ground rule:

$$\{H_1, \dots, H_n\} \leftarrow C_1, \dots, C_n, \text{not } C_{n+1}, \dots, \text{not } C_{n+m}$$

where H_i are basic atoms and C_i are cardinality atoms. Then, for each cardinality atom C_i we create a new ground atom c_i , and replace R with the $n + m + 1$ *smodels* rules:

$$\begin{aligned} \{H_1, \dots, H_n\} &\leftarrow c_1, \dots, c_n, \text{not } c_{n+1}, \dots, \text{not } c_{n+m} \\ c_1 &\leftarrow C_1 \\ &\vdots \\ c_{n+m} &\leftarrow C_{n+m} . \end{aligned}$$

9 PROGRAMMING METHODOLOGY

In this chapter we examine some general principles that we can use to construct cardinality constraint logic programs. Most of the ideas can be applied to other answer set formalisms with little modification.

In general, CCPs are most suitable for modeling problems whose decision versions are **NP**-complete. If a problem has an efficient polynomial-time algorithm, then expressing it using an **NP**-complete formalism such as ground CCPs may be counter-productive. Similarly, if the problem falls outside **NP**, solving it with a single program often results in a complex and unintuitive program.¹ From a practical viewpoint it is often simpler to solve the problem using an algorithm that includes a CCP-solver as an **NP**-oracle [200, 152]. For example, the **PSPACE**-complete problem of finding an optimal plan can be reduced into solving a series of **NP**-complete problems of finding a plan of a specific length.

9.1 GENERATE AND TEST METHOD

Writing answer set programs using the *generate and test* method [141, 60] makes them often easier to create, understand, and debug. In ASP context it means that we divide our problem into two parts:

1. a *generator* that creates all possible solution candidates; and
2. a *tester* that checks whether a particular candidate is, indeed, a valid solution.

How we write the generator depends on what ASP semantics we are using. With cardinality constraint programs the natural way is to use choice rules.

In the very simplest form an encoding has two rules. One choice rule that allows us to choose a set of atoms into the answer set, and one constraint that rejects those solutions that are not valid. These programs have the general form of:

$$\begin{aligned} &\{X.choose(X) : option(X)\} \leftarrow \\ &\leftarrow choose(X), invalid-choice(X) . \end{aligned}$$

We will now examine how we can express the well-known graph vertex coloring problem with this approach.

PROBLEM 5: VERTEX COLORING. Given a graph $G = \langle V, E \rangle$ and a set of colors C , is it possible to assign a color for each vertex such that all adjacent vertices have different colors?

¹With the exception that many Σ_P^2 -complete problems have also natural ASP encodings when we use a formalism based on disjunctive logic programs [37].

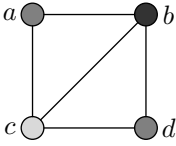
Graph	Facts	Stable Model
	$vtx(a); vtx(b); vtx(c); vtx(d);$ $edge(a, b); edge(a, c); edge(b, c);$ $edge(b, d); edge(c, d); color(r);$ $color(g); color(b)$	$\{has-color(a, g),$ $has-color(b, r),$ $has-color(c, b),$ $has-color(d, g)\}$

Figure 9.1: Example of VERTEX COLORING.

Example 9.1.1 We can encode VERTEX COLORING with the following program:

$$\begin{aligned}
&1 \{C.has-color(V, C) : color(C)\} \quad 1 \leftarrow vtx(V) \\
&\leftarrow edge(X, Y), has-color(X, C), has-color(Y, C) .
\end{aligned}$$

The first rule is the generator that creates answer set candidates where every vertex is colored with one color. The second rule tests that no adjacent nodes have the same color.

The predicate symbols $vtx/1$, $edge/2$, and $color/1$ are part of the input and they define the particular problem instance that we are trying to solve. Figure 9.1 shows an example of finding a 3-coloring for a four vertex graph.

In more involved examples both the generator and the tester can be more complex. We then include basic rules that define the necessary auxiliary predicates. There are two main uses for the auxiliaries:

1. definition of the tester rules can be difficult or even impossible without them; and
2. they allow us to have tighter definitions for the generator.

For example, when we defined the Hamiltonian cycle encoding in Example 4.1.2, we had to define the reachability predicate $r/1$ because we needed it to check whether the cycle visits every node.

The second reason is closely related to the search spaces of the program. The fewer answer set candidates that we can derive with rules of the generator, the smaller the search space is.

9.2 UNIFORM ENCODINGS

A problem encoding is *uniform* [171, 60, 130] if we can use it to solve all instances of the problem: we have a non-ground program P that encodes the constraints of the problem, and each problem instance is defined as a set of ground facts. We can form uniform ASP encodings for all **NP** search problems.

The main advantage of using a uniform encoding is *modularity*. We can often create a program by partitioning it into subproblems and then combining their encodings. This is much easier to do when the encodings are uniform and then we can also replace a component by another logically equivalent one.

It is customary to divide the predicate symbols that occur in a program into two classes: *data* and *program predicates* [26]. The data predicates are fixed and are given as input to the program in the form of facts while program predicates act upon the data. This division is not the same as the division between domain and non-domain predicates since we will often define additional domain predicates in terms of data predicates.

When we have a uniform encoding for a problem P , we use $P(I)$ to denote the program that we get from combining the facts that define an instance I of P to the encoding. The encodings will often have parameters that are defined using interpreted function symbols. In these cases we add the parameters to the program name so that $P(I, k)$ denotes the program for the instance I where a numeric parameter has the value k .

9.2.1 Examples

We now give several examples for uniform encodings of problems. We have already seen a couple of them: the encoding for VERTEX COLORING in Example 9.1.1, the HAMILTONIAN CYCLE encoding in Example 4.1.2, and the Turing machine simulation in Section 6.4.

Next, we examine three different variants of the propositional logic satisfiability problem, SAT, BOOLEAN SAT, and MAXSAT. In the first variant we examine the satisfiability of a set of clauses, then we extend it to cover arbitrary boolean formulas, and finally we examine the problem of finding a truth assignment that satisfies the greatest number of clauses in a possibly inconsistent clause set.

Sat

Since the satisfaction problem (SAT) of propositional logic is the canonical example of **NP**-complete problems, we examine its encoding in detail.

PROBLEM 6: SAT. Let F be a propositional logic formula in conjunctive normal form. The problem is to determine whether F satisfiable.

We suppose that F is given as a set of facts of the form $atom(a)$, $clause(c)$, $pos(c, a)$ and $neg(c, b)$. The first two predicates define what atoms and clauses there are in the formula, while the next two encode what positive and negative literals belong to the clauses.

ENCODING 6: SAT.

$$\{A.true(A) : atom(A)\} \leftarrow \quad (1)$$

$$sat(C) \leftarrow clause(C), pos(C, A), true(A) \quad (2)$$

$$sat(C) \leftarrow clause(C), neg(C, B), not\ true(B) \quad (3)$$

$$\leftarrow clause(C), not\ sat(C) \ . \quad (4)$$

The rule (1) creates all possible truth assignments for the atoms, while (2) and (3) assert that a clause is satisfied if some literal in it is true. Finally, (4) states that it is an error if some clause is not satisfied.

Now, we argue that this translation is correct and that the program $\text{SAT}(F)$ that we get by adding the instance as facts to (1)–(4) has a stable model M if and only if the set of clauses F has a satisfying truth assignment \mathcal{V} where $\mathcal{V} = \{A \mid \text{true}(A) \in M\}$. This is an example how we can prove problem encodings correct. We will not give proofs for the other problems in this section.

Proposition 9.2.1 *Let F be a propositional logic formula in a conjunctive normal form. Then F is satisfiable if and only if the program $\text{SAT}(F)$ has a stable model.*

Proof. First, suppose that $\text{SAT}(F)$ has a stable model M and consider the set of propositional atoms $\mathcal{V} = \{a \mid \text{true}(a) \in M\}$. Rule (4) of $\text{SAT}(F)$ demands that the atom $\text{sat}(c) \in M$ for every clause c . By Theorem 3.6.2 each such an atom has to have a justification in M . There are two ways to derive $\text{sat}(c)$:

1. there is a propositional atom a such that $\text{pos}(c, a)$ and $\text{true}(a)$ are true in M ; or
2. there is a propositional atom b such that $\text{neg}(c, b) \in M$ but $\text{true}(b) \notin M$.

In the first case c has a positive literal a and $a \in \mathcal{V}$ so $\mathcal{V} \models c$. Similarly, in the second case c has a negative literal $\neg b$ and $b \notin \mathcal{V}$ so $\mathcal{V} \models c$. Thus, \mathcal{V} is a model of F .

Conversely, suppose that \mathcal{V} is a model of F and let $I(F)$ denote the set of instance atoms:

$$I(F) = \{p(t_1, \dots, t_n) \mid p \in \{\text{clause}, \text{pos}, \text{neg}, \text{atom}\} \text{ and } \langle p(t_1, \dots, t_n), \emptyset \rangle \in \text{SAT}(F)\}.$$

Consider the set M of ground atoms:

$$M = \{\text{true}(a) \mid a \in \mathcal{V}\} \cup \{\text{sat}(c) \mid \text{clause}(c) \in I(F)\} \cup I(F)$$

We now examine the reduct $\text{inst}_{\mathbf{H}}(P(F))^M$. First, note that all atoms in $I(F)$ were defined with ground facts so its atoms are trivially in the least model of the reduct.

Next, the reduct of the rule (1) is the set of facts:

$$\{\langle \text{true}(a), \emptyset \rangle \mid a \in \mathcal{V}\}$$

and the reduct of (3) is the set of rules:

$$\{\langle \text{sat}(c), \{\text{clause}(c), \text{neg}(c, b)\} \rangle \mid b \notin \mathcal{V}\}$$

while the reducts of the remaining rules contain all possible variable substitutions.

Consider a clause c . Since \mathcal{V} is a model of F , there is a literal $l \in c$ such that $\mathcal{V} \models l$. If $l = \text{not } b$ is negative, then $b \notin \mathcal{V}$ so the reduct contains a rule:

$$\text{sat}(c) \leftarrow \text{clause}(c), \text{neg}(c, b)$$

and we get $sat(c)$ into the least model.

If $l = A$ is positive, then we have a rule:

$$sat(c) \leftarrow clause(c), pos(c, a), true(a)$$

whose body is satisfied in the least model since $true(a)$ is true as $a \in \mathcal{V}$ and we again get $sat(c)$ into the least model. Thus,

$$\mathbf{MM}(\text{inst}_{\mathbf{H}}(P(F))^M) = M$$

so M is a stable model of $\text{SAT}(F)$. □

Boolean Sat

In SAT we examined only propositional logic formulas that were in conjunctive normal form. Now we generalize this to boolean formulas that are formed with the full set of standard connectives: disjunction (\vee), conjunction (\wedge), negation (\neg), implication (\rightarrow), and equivalence (\leftrightarrow).

PROBLEM 7: BOOLEAN SAT. Let F be a propositional logic formula. The problem is to find if F satisfiable.

We construct the encoding by breaking F into its subformulas and defining a new atom for each such a formula. We do this using a function $t(F)$ that translates a boolean formula into a set of facts. During the translation we assign a unique identifier $i(F)$ for every subformula.

Definition 9.2.1 *Let F be a boolean formula, then the translation $t(F)$ is defined as follows.*

1. *If $F = A$ for some propositional atom A , then*

$$t(A) = \{atom(i(A)) \leftarrow\}$$

where $i(A)$ is a unique identifier for the atom A .

2. *If $F = \neg\alpha$, then*

$$t(\neg\alpha) = \{negate(i(F), i(\alpha)) \leftarrow\} \cup t(\alpha)$$

where $i(F)$ and $i(\alpha)$ are the identifiers of the corresponding subformulas.

3. *If $F = \alpha \wedge \beta$, then*

$$t(F) = \{and(i(F), i(\alpha), i(\beta)) \leftarrow\} \cup t(\alpha) \cup t(\beta) .$$

4. *If $F = \alpha \vee \beta$, then*

$$t(F) = \{or(i(F), i(\alpha), i(\beta)) \leftarrow\} \cup t(\alpha) \cup t(\beta) .$$

5. *If $F = \alpha \rightarrow \beta$, then*

$$t(F) = \{implies(i(F), i(\alpha), i(\beta)) \leftarrow\} \cup t(\alpha) \cup t(\beta) .$$

6. If $F = \alpha \leftrightarrow \beta$, then

$$t(F) = \{equal(i(F), i(\alpha), i(\beta)) \leftarrow\} \cup t(\alpha) \cup t(\beta) .$$

In addition to the facts $t(F)$, we need to specify which subformula corresponds to the complete formula F . For this we add an extra fact $formula(i(F)) \leftarrow$.

Example 9.2.1 Let $F = (A \vee \neg B) \wedge (B \rightarrow C)$. Then, the translation $t(F)$ is the set of facts:

$$\begin{array}{ll} atom(a) \leftarrow & negate(1, b) \leftarrow \\ atom(b) \leftarrow & or(2, a, 1) \leftarrow \\ atom(c) \leftarrow & implies(3, b, c) \leftarrow \\ and(4, 2, 3) \leftarrow & formula(4) \leftarrow \end{array}$$

We first assign truth values to all propositional atoms using the same method as in the previous example, and then we propagate the truth values to compute the truth valuations of the subformulas of the instance.

ENCODING 7: BOOLEAN SAT.

$$\begin{array}{l} \{A.true(A) : atom(A)\} \leftarrow \\ true(F) \leftarrow negate(F, F_1), not\ true(F_1) \\ true(F) \leftarrow and(F, F_1, F_2), true(F_1), true(F_2) \\ true(F) \leftarrow or(F, F_1, F_2), true(F_1) \\ true(F) \leftarrow or(F, F_1, F_2), true(F_2) \\ true(F) \leftarrow implies(F, F_1, F_2), not\ true(F_1) \\ true(F) \leftarrow implies(F, F_1, F_2), true(F_2) \\ true(F) \leftarrow equal(F, F_1, F_2), true(F_1), true(F_2) \\ true(F) \leftarrow equal(F, F_1, F_2), not\ true(F_1), not\ true(F_2) \\ \leftarrow formula(F), not\ true(F) \end{array}$$

Now, if F is satisfiable, then its encoding has a stable model M and the atoms $true(f)$ give us a valuation \mathcal{V} that satisfies F . In particular, for all atoms a occurring in F it holds that $\mathcal{V}(a) = T$ if and only if $true(i(a)) \in M$. Similarly, if M is a stable model of the encoding, it corresponds to a model of F .

Maxsat

PROBLEM 8: MAXSAT. Let C be a set of clauses and k be an integer. Is there a truth assignment that satisfies at least k clauses?

This problem can be solved using a program that is essentially equivalent to the one for SAT with the exception that instead of having a constraint to weed out assignments that are not models, we define an upper limit k and say that it is an error if less than k clauses are satisfied.


```

function oracle(Program  $P$ )
    if  $P$  has an answer set  $M$  then
        return  $\langle \text{sat}, M \rangle$ 
    else
        return  $\langle \text{unsat}, \emptyset \rangle$ 
    endif
endfunction

```

Figure 9.2: The basic oracle algorithm

ENCODING 8: MAXSAT.

$$\begin{aligned}
 \{A.\text{true}(A) : \text{atom}(A)\} &\leftarrow \\
 \text{sat}(C) &\leftarrow \text{clause}(C), \text{pos}(C, A), \text{true}(A) \\
 \text{sat}(C) &\leftarrow \text{clause}(C), \text{neg}(C, B), \text{not true}(B) \\
 &\leftarrow \{C.\text{sat}(C) : \text{clause}(C)\} \quad k - 1 \ .
 \end{aligned}$$

The program $\text{MAXSAT}(F, k)$ has a stable model M if and only if F has a truth assignment that satisfies at least k clauses. The satisfying truth assignment is the set of atoms $\{A \mid \text{atom}(A) \in M\}$.

In the next section we see how we can solve the functional variant of this problem using the oracle method.

9.3 USING ANSWER SET SOLVERS AS ORACLES

The answer set formalisms that are in common use are not Turing-complete. This means that we cannot use them for general programming. When we want to use ASP as a part of an application, we do it by embedding an ASP solver into the application and calling it from the program to create the answer sets that we want.

When designing the algorithms that make use of the ASP solver we can think that the solver takes the role of an *oracle* [200, 152]. In the theory of computation an oracle is a hypothetical machine that can solve some particular problem. We encode the interesting problem as an ASP program, and then use the solver as black box that gives us one or more answer sets of the program.

Basic Oracle

Figure 9.2 shows the basic form of an oracle. We give an ASP program to the oracle, and it then returns a pair $\langle R, S \rangle$ where $R \in \{\text{sat}, \text{unsat}\}$ tells whether the program has an answer set at all and S is one of its answer sets if they exist. If there are no answer sets at all, we set $S = \emptyset$.

Using the Oracle

Figure 9.3 shows the general form how we can use the oracle. If we want to solve a problem P , we start by writing a uniform encoding P for it. Then, we write a function $\text{create-facts}(I)$ that takes as its input

```

function solve-problem(Instance  $I$ )
    Let  $P$  be a uniform encoding for the problem
     $P_I := \text{create-facts}(I)$ 
     $\langle R, M \rangle := \text{oracle}(P \cup P_I)$ 
    if  $R = \text{unsat}$  then
        return unsat
    else
        return interpret( $M$ )
    endif
endfunction

```

Figure 9.3: The general form for using an oracle

```

function functional-maxsat(ClauseSet  $F$ )
    Let  $P(k)$  be the MAXSAT encoding of  $F$ 
    upper :=  $|F|$ 
    lower := 0
    while lower < upper do
         $k := \lceil (\text{upper} - \text{lower}) / 2 \rceil$ 
         $\langle R, M \rangle = \text{oracle}(P(k))$ 
        if  $R = \text{sat}$  then
            lower :=  $k$ 
        else
            upper =  $k - 1$ 
        endif
    endwhile
    return lower
endfunction

```

Figure 9.4: An algorithm for computing functional MAXSAT

an instance I of P and it returns a set of facts P_I that corresponds to I . We then call the oracle with the program $P \cup P_I$ to find an answer set M . Finally, we extract the answer from M using another function $\text{interpret}(M)$ that translates the answer back to the terms of our original problem.

Example 9.3.1 *In the functional version of MAXSAT (Problem 8) we have a set of clauses F and we want to find out how many of them we can satisfy at the same time. We see an algorithm to compute this Figure 9.4. We do a binary search over the possible values of k and then return the largest value that left us with a satisfiable program.*

Oracle with Exclusion

We may want to use the oracle to find answer sets that have some specific properties. We can do it by modifying the encoding to reject answer set candidates that do not have the properties.

In the simplest case we have some set $M = \{L_1, \dots, L_n\}$ of basic

```

function oraclewe(Program  $P$ , Exclusions  $E$ )
     $P' := P \cup \{\langle \perp, M \rangle \mid M \in E\}$ 
    return oracle( $P'$ )
endfunction

```

Figure 9.5: Oracle with exclusion sets

```

function all-answer-sets(Program  $P$ )
     $\mathbf{SM} := \emptyset$ 
     $\langle R, M \rangle := \text{oracle}(P)$ 
    while  $R \neq \text{unsat}$  do
         $M' = M \cup \{\text{not } A \mid A \in \text{Atoms}(P) \setminus M\}$ 
         $\mathbf{SM} := \mathbf{SM} \cup \{M'\}$ 
         $\langle R, M \rangle := \text{oracle}_{\text{we}}(M, \mathbf{SM})$ 
    endwhile
    return  $\mathbf{SM}$ 
endfunction

```

Figure 9.6: A naive way of computing all answer sets

literals that we do not want to be true at the same time. We can reject the undesired answer sets by adding a rule:

$$\leftarrow L_1, \dots, L_n$$

to the program. In Figure 9.5 we see an algorithm $\text{oracle}_{\text{we}}$ that generalizes this to a set of sets of literals to exclude.

We can use $\text{oracle}_{\text{we}}$ to compute all answer sets of a program. A naive algorithm for this is shown in Figure 9.6. The idea is that we compute the answers sets one at a time, and each time we find one we add it to the set of exclusions. In practice, this algorithm has two major weaknesses. First, it always starts the answer set computation from the beginning. Most ASP systems have an option to compute all answer sets and it is usually more efficient to use that directly. The second weakness is that a program can have an exponential number of answer sets. This leads to a worst case exponential space use in the algorithm.

Two-Program Oracles

As our final example we consider the interleaved two-program construction [102, 13] that allows us to use **NP**-encodings for Σ_P^2 problems.²

As an example we encode the Σ_P^2 -complete problem of deciding whether a 2-quantified boolean formula is satisfiable [152] using this method.³

PROBLEM 9: 2-QUANTIFIED BOOLEAN FORMULAS. A 2-quantified boolean formula (QBF₂) has the form:

$$\exists X_1 \forall X_2. f(X_1, X_2)$$

²The class Σ_P^2 contains problems that can be solved in a nondeterministic polynomial time if we have an access to an **NP**-oracle.

³Numerous direct algorithms for solving QBFs have been presented in literature [17].

```

function QBF(Formula F, Variables  $X_1$ )
  Let  $P_1 := \text{BOOLEAN SAT}(F)$ 
  Let  $P_2 := \text{BOOLEAN SAT}(\neg F)$ 
   $E := \emptyset$ 
   $\langle R, M \rangle := \text{oracle}(P_1)$ 
  while  $R \neq \text{unsat}$  do
     $E := E \cup M$ 
     $P' := P_2 \cup \{ \langle \text{true}(a), \emptyset \rangle \mid a \in M \cap X_1 \} \cup \{ \langle \perp, \{ \text{true}(a) \} \rangle \mid a \in X_1 \setminus M \}$ 
     $\langle R', M' \rangle := \text{oracle}(P')$ 
    if  $R' = \text{unsat}$  then
      return  $\langle \text{sat}, M \cap X_1 \rangle$ 
    endif
     $\langle R, M \rangle := \text{oracle}_{\text{we}}(P_1, E)$ 
  endwhile
  return  $\langle \text{unsat}, \emptyset \rangle$ 
endfunction

```

Figure 9.7: Solving 2-Quantified Boolean Formulas

where $f(X_1, X_2)$ is a boolean formula over two sets of propositional atoms X_1 and X_2 . The problem is to decide whether there exists a truth assignment for atoms in X_1 such that $f(X_1, X_2)$ is true for all truth assignments of X_2 .

Example 9.3.2 Suppose that we have a 2-QBF $Q = \exists X_1 \forall X_2. f(X_1, X_2)$ where $X_1 = \{x_1, \dots, x_n\}$. The key for solving the problem is to note that a truth assignment $\sigma = \{x_1/v_1, \dots, x_n/v_n\}$ fulfills the conditions of the definition if

1. $f(X_1, X_2)\sigma$ is satisfiable; but
2. $\neg f(X_1, X_2)\sigma$ is unsatisfiable.

This gives us the algorithm shown in Figure 9.7: we first find a classical model M for $f(X_1, X_2)$. Then, we add new rules to the encoding of $\neg f(X_1, X_2)$ to fix the values of X_1 to agree with M . For each atom $a \in M \cap X_1$ we add the rule $\text{true}(a) \leftarrow$ and for the atoms $a \in X_1 \setminus M$ we add a constraint $\perp \leftarrow \text{true}(a)$. If the resulting program has no answer sets, then $\neg f(X_1, X_2)\sigma$ is not satisfiable, so we have found a model for Q . If it has an answer set, we go on to examine the next model candidate.

This same idea can be used for all Σ_P^2 problems. We create two programs P_1 and P_2 , and the problem has a solution if P_1 has an answer set but P_2 has not. Before we call the solver for P_2 we modify it to take into account the answer set that we found in the earlier step.

9.4 ON OPTIMIZATION

It is very common that two different encodings for a problem that are logically equivalent have vastly different performance characteristics in

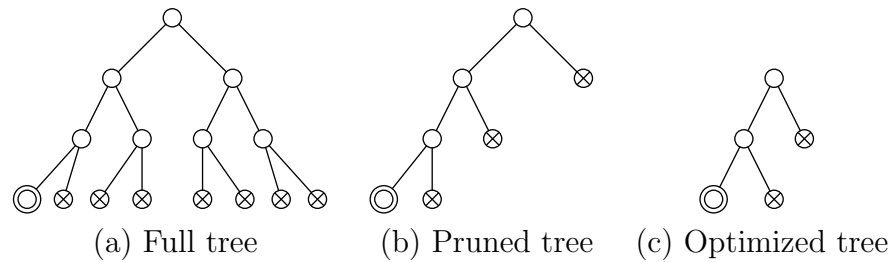


Figure 9.8: Sample search trees

actual use.

Computing answer sets of a program is a difficult operation because we are usually working on **NP**-hard problems. In difficult problem instances the solver can make a poor choice early in the computation and then spend a lot of time examining a part of the search space that does not contain any answer sets at all [93].

Consider the search tree shown in Figure 9.8a. There the idea is that the program has only one answer set that is found at the very leftmost part of the tree while all other branches end in contradictions. If the solver chooses to first explore the right branch, then it has to do a lot of futile work before it can backtrack to the correct branch.

We can often improve the performance of the solver by adding some extra constraints that prune out whole branches of the search tree. Figure 9.8b illustrates what happens when we add a constraint that immediately causes a contradiction whenever we take a right branch. The resulting tree is significantly smaller than the full tree.

Every ground atom that we have in the program can in the worst case double the size of the search. This makes it worthwhile to create the encoding in a way that the size of its non-domain Herbrand base is as small as possible. Figure 9.8c shows how get even smaller tree when we can remove one of the atoms from the program.

At this point there are no comprehensive guides for optimizing answer set programs. It is likely that there is no single approach that always would lead to the most efficient encoding. For example, it is possible that sometimes adding a new atom may allow us to define new constraints to prune out the search space so we get a more efficient encoding even though its Herbrand base is larger. It can also happen that adding a new constraint increases the running time because processing it takes some time and it removes so small portions of the search space that it could have been explored faster.

Our practical experience has been that using the following procedure leads to reasonably efficient encodings:

1. Create a working encoding in the most straightforward way as possible.
2. Define new domain predicates that allow tighter definitions for rules in the generator in the sense that we get less candidate answer sets.
3. Add new constraints to prune out parts of search space that can be guaranteed to not contain valid solutions.

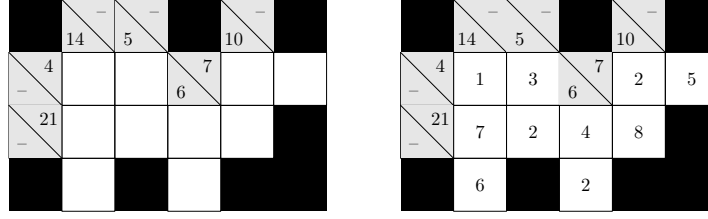


Figure 9.9: A cross-sum/Kakuro puzzle and a solution

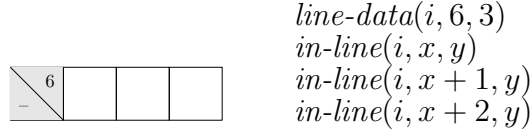


Figure 9.10: Kakuro input encoding

9.4.1 Optimization Example: Kakuro Puzzles

We now examine how we could develop iteratively an encoding for kakuro (cross sum) puzzles. In a kakuro puzzle we have to find a way to place a number from the interval 1–9 to each square of a grid so that the rows and columns sum to given totals. There is an additional restriction that each total may contain each number at most one time. Figure 9.9 shows an example puzzle and one of its solutions.⁴

The first question is how to represent the puzzle. We take here the simple way out and suppose that the lines and what squares belong to them are already identified in the input. Our actual implementation combines a Perl program with and a CCP to compute the line data.

We assign a unique identifier l for every line and then use the predicates $line-data/3$ and $in-line/3$ to identify the line totals and which squares belong to which lines. Figure 9.10 shows an one example line and its encoding as facts. Here x and y are the coordinates of the leftmost square.

Direct Encoding

In the straightforward encoding we first choose a number for every square of the grid:

$$1 \{N.has-number(N, X, Y) : number(N)\} \quad 1 \leftarrow square(X, Y) \quad .$$

Next, it is an error if the sum of a line is too big. We use a weight constraint to compute the sums:

$$\begin{aligned} \leftarrow T + 1 \quad & \{ \{N, X, Y\}.has-number(N, X, Y) : \\ & number(N) \wedge in-line(I, X, Y) = N \}, \\ & line-data(I, T, L) \quad . \end{aligned}$$

⁴This is a poor quality example since it has more than one solution. A well-formed kakuro puzzle has a unique answer that can be found without having to resort to guessing.

It is also an error if a line has a too small sum:

$$\begin{aligned} \leftarrow [\{N, X, Y\}.has-number(N, X, Y) : \\ number(N) \wedge in-line(I, X, Y) = N] T - 1, \\ line-data(I, T, Len) . \end{aligned}$$

Finally, no line can contain the same number two times:

$$\begin{aligned} \leftarrow 2 \{ \{X, Y\}.has-number(N, X, Y) : in-line(I, X, Y) \}, \\ number(N), line(I) . \end{aligned}$$

Optimized Encoding

We can improve this encoding by noticing that we are generating a great number of spurious choices. Consider the line from Figure 9.10. The line is three squares long and its total is 6. The only combination of three different numbers with that sum is $1 + 2 + 3 = 6$ so we know that the squares contain some permutation of those numbers. However, our encoding contains rules for placing all numbers to them. The constraints prune out immediately the model candidates with too large numbers, but it adds a small amount of unnecessary work for the solver.

We can compute all possible sums in advance and make the choice over only the numbers that can occur in combinations that fit in a given line. Since each non-zero digit may occur zero or one times in a line, we have $2^9 - 1 = 511$ possible combinations. We use the predicates *combination(id, sum, length)* and *in-combination(id, n)* to define them. For example, the example of $1 + 2 + 3$ would be represented with:

$$\begin{aligned} combination(id, 6, 3) \leftarrow \\ in-combination(id, 1) \leftarrow \\ in-combination(id, 2) \leftarrow \\ in-combination(id, 3) \leftarrow . \end{aligned}$$

Since the combinations are always the same, we add them as facts to the problem encoding.

We start by choosing which combinations can occur in a given line:

$$p-combination(I, C) \leftarrow line-data(I, T, L), combination(C, T, L) .$$

Each number that belongs to a possible combination can occur in any of the squares of the line:

$$\begin{aligned} p-number(N, X, Y) \leftarrow in-line(I, X, Y), p-combination(I, C), \\ in-combination(C, N) . \end{aligned}$$

We choose one combination for every line:

$$1 \{ C.has-combination(I, C) : p-combination(I, C) \} 1 \leftarrow line(I)$$

and one number for every square:

$$1 \{ N.has-number(N, X, Y) : p-number(N, X, Y) \} 1 \leftarrow square(X, Y) .$$

Encoding	Small		Large	
	Atoms	Rules	Atoms	Rules
Direct				
Complete	252	275	11420	12629
Non-domain	195	211	9041	10250
Optimized				
Complete	3478	4116	25051	30218
Non-domain	185	605	6576	9965

Table 9.1: Instantiation sizes for the Kakuro examples

We have to also ensure that the number that we chose actually belongs to the correct combinations:

$$\leftarrow \text{has-number}(N, X, Y), \text{has-combination}(I, C), \\ \text{in-line}(I, X, Y), \text{not in-combination}(C, N) .$$

Finally, we have to check that one number does not occur twice in a line:

$$\leftarrow 2 \{ \text{has-number}(N, X, Y) : \text{in-line}(I, X, Y) \}, \\ \text{p-combination}(I, C), \text{in-combination}(C, N) .$$

Comparing Encodings

The Tables 9.1 and 9.2 show how the two encodings compare with each other in case of two examples. The small example is the one given in Figure 9.9 and the large example is a huge puzzle with 350 lines and 880 squares that was that was created by Paul A. Grosse⁵

When we examine the sizes of the ground instantiation we consider two cases: whether we include the complete relevant instantiation or only the non-domain part. Since computing the domain model is straightforward and we already need to do it to create the relevant instantiation, we usually leave the domain program out when we finally call an ASP solver to compute the answer sets.

Instance Sizes The first thing to note in Table 9.1 is that in the case of optimized encoding the size of the domain program is much larger than the size of the non-domain program. This is because we compute the set of numbers that can occur in a given square as a part of the domain program.

In both examples the non-domain part of the optimized encoding has fewer atoms. This is as expected as now we have there only those number choices that can fit in a given place instead of trying to place any number there. In the small example the direct encoding has marginally fewer rules, but in the large one we see the amount of savings that we get from the optimizations.

⁵Puzzle identifier 20060327, downloaded from <http://www.grosse.is-a-geek.com/kakbig.html> .

Encoding	System	One Solution		All Solutions	
		Small	Large	Small	Large
Direct	<i>smodels</i>	2	595	13	992
	clasp	32	27465	160	27484
	clasp-lh	24	7248	91	7256
Optimized	<i>smodels</i>	3	0	10	0
	clasp	25	106	59	151
	clasp-lh	2	0	10	0

Table 9.2: Search tree sizes of Kakuro examples with *smodels*-2.32 and clasp-1.1.0. For clasp the figures are with and without lookahead but otherwise it was run with the default options.

	-	-		-	
	14	5		10	
4			7		3
-			6		
21			4		
-					

Figure 9.11: Two choices that fix a unique solution.

Search Tree Sizes The Table 9.2 compares the search trees for the encodings that were computed using the *smodels* [174] and clasp [79] solvers.

Smodels combines a DPLL-style [39] branching search with an extensive set of propagation rules that are described in detail in [174]. Clasp extends branching search with conflict-driven clause learning. For clasp we tested the cases with and without the lookahead heuristics.

There are eleven correct solutions for the small example. The least number of choices that we have to make before the values of the other squares are determined is two. Figure 9.11 shows one pair of two numbers that together fix the other numbers.

We found a solution with an optimal number of choices in two cases: using *smodels* with the direct encoding and using clasp with lookahead on the optimized encoding. Here *smodels* made one unnecessary choice with the optimized encoding. This is probably because adding the new rules for choosing the combinations of numbers misled the heuristics.

When we wanted to find all valid solutions, the optimized encoding was strictly better in all test cases. With the optimized encoding both *smodels* and clasp with lookahead found the solutions in an optimal number of choices. Both solvers had to explore unsatisfiable parts of the search tree when the direct encoding was used.

The advantages of optimization become clear with the large example. The large example has only one solution and both *smodels* and clasp with lookahead could find it without making any choices at all, while clasp without lookahead needed a bit over 100 choices. With the direct

encoding *smodels* needed almost 600 choices, clasp with lookahead needed 7,000, and clasp without lookahead almost 30,000. This example shows that there can be very large differences in search spaces when we use different encodings for a problem.

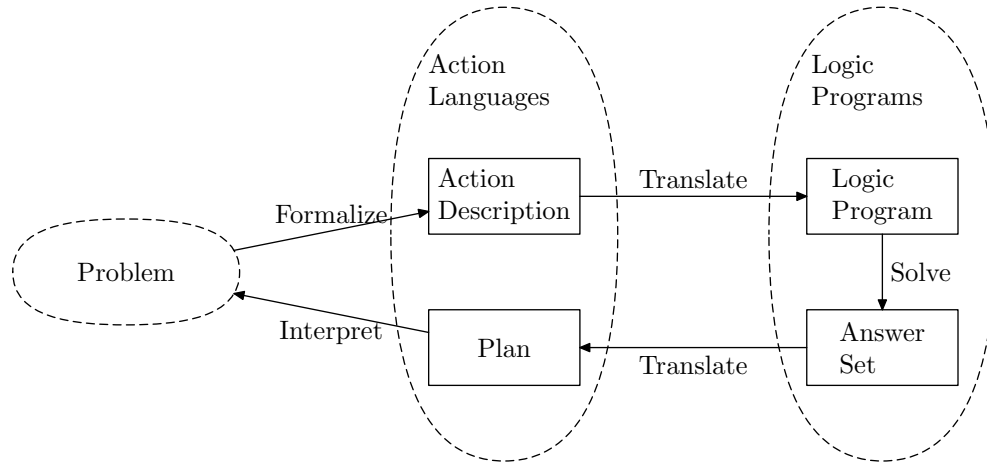


Figure 10.1: Solving planning problems

10 ENCODING PLANNING PROBLEMS

In this chapter we examine in detail how we can model planning problems using cardinality constraint programs. In a planning problem we have to find a way to reach a goal. For example, the problem might be that we have a set of packages that we have to deliver to different addresses in a city and we want to find out the fastest way to deliver them to the correct destinations.

Much work has been published on answer set planning [87, 114, 115, 47, 57, 55, 54, 48, 104, 173, 156, 23, 136, 57, 49]. We will here show a general set of techniques that can then be applied to automatically translate higher-level description languages into cardinality constraint programs. This is not a new idea by itself and it is exactly what most ASP planners do in practice. We present here a translation that uses the features of cardinality constraint programs. Our approach is similar to the translation to disjunctive logic programs that is used in the DLV^K planning system [57, 56].

10.1 GENERAL APPROACH

When we want to solve a real-world planning problem, our first step is to abstract it to the level that we can express it with a suitable planning definition formalism. In this work we use *action language* [135] based formalisms for this.

Next, we translate the action language description into a cardinality constraint program and use an answer set solver to find an answer set for it. From this answer set we extract the atoms that correspond to the plan and translate them back into action language terms.

Finally, we examine what real-world actions we have to take to execute the plan. This process is shown in Figure 10.1.

10.2 DIFFERENT FORMS OF PLANNING

There are many different forms of planning problems. The first major division is between *sequential* and *parallel* planning. In sequential planning we can make only one operation at the time while in parallel planning we can make multiple actions at the same time as long as they are not mutually incompatible. There are several possible ways how we can define this incompatibility, and we will examine more about them later.

The second division is *temporal* and *non-temporal*. In non-temporal planning each action takes the same time to execute no matter how complex it is. In temporal planning the actions can take different durations to execute.

Third division is between *static* and *dynamic* domains. In static domains only the operations that are taken can change the world state, but in dynamic domains the world may change independently from the actions taken.

The fourth thing to consider is whether the domain is *complete* or *incomplete* knowledge world. In complete knowledge domains we can access the values of all state predicates all time during the planning. If the values of some predicates are unknown, then it is an incomplete knowledge planning domain.

The fifth division is whether the actions are *conditional* or *unconditional*. An action is conditional if it can have different effects based on the state of the world when it is executed. If we additionally have an incomplete world, then we can say that a conditional action is *nondeterministic* as we might not know what effect the action will have.

The sixth division is whether the actions are *weighted* or *unweighted*. In weighted planning each action has a different cost associated to it and we want to find a plan with smallest possible total cost.

In this work we concentrate on parallel planning in non-temporal static worlds with complete knowledge but we make small excursions to other forms of planning.

10.3 FORMALIZING PLANNING

Planning domains are usually defined using either some derivative of STRIPS [73] planning domain description language such as PDDL [137] or an action language [135, 88, 92, 91, 57]. In answer set planning this description is then translated—either by hand or automatically—to an ASP program that is then used to find the actual plans.

We use the action language \mathcal{B} [88] as our underlying formalism. We select this language because it is the simplest action language that includes static laws. We want the basic language to be simple so that the translation into CCPs is straightforward and we want to include static laws because they are very convenient in modeling planning domains. We will examine some aspects of more complex action languages later in Section 10.8.

10.3.1 The Action Language \mathcal{B}

There are two different kinds of entities in an action language: *actions* and *fluents*. Fluents encode the state of the world and actions describe how we can change it.

Definition 10.3.1 An action signature $\mathcal{S} = \langle \mathcal{A}, \mathcal{F} \rangle$ consists of a set \mathcal{A} of elementary actions and a set \mathcal{F} of fluents.

An action description over an action signature contains *dynamic* and *static laws*. A dynamic law describes how an action works, and a static law describes a condition that has to hold in every state of the planning domain.

Definition 10.3.2 (\mathcal{B}) An action description $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ is a triple consisting of an action signature $\mathcal{S} = \langle \mathcal{A}, \mathcal{F} \rangle$, a set \mathcal{L}_s of static laws, and a set \mathcal{L}_d of dynamic laws, where a static law is of the form:

$$L \text{ if } F$$

where L is a literal and F is a conjunction of literals that are formed using fluents from \mathcal{F} , and a dynamic law is of the form:

$$A \text{ causes } L \text{ if } F$$

where $A \in \mathcal{A}$, L is a literal, F is a conjunction of literals formed using fluents from \mathcal{F} .

We use two special fluents in our action description, \top and \perp where \top is a fluent that is always true and \perp is always false. We treat them as if we had the following two static laws defined for them:

$$\begin{aligned} &\top \\ &\perp \text{ if } \neg \top . \end{aligned}$$

Furthermore, when we have a construct of the form $L \text{ if } \top$, we leave the conditional part out. For example, $A \text{ causes } L \text{ if } \top$ is written simply as $A \text{ causes } L$.

Definition 10.3.3 Let $\mathcal{L} = A \text{ causes } L \text{ if } F_1 \wedge \dots \wedge F_n$ be a dynamic law, then its effect $\text{eff}(\mathcal{L})$ and conditions $\text{cond}(\mathcal{L})$ are defined as follows:

$$\begin{aligned} \text{eff}(\mathcal{L}) &= L \\ \text{cond}(\mathcal{L}) &= \{F_1, \dots, F_n\} . \end{aligned}$$

A planning instance consists of an initial state and a goal condition. We start from the initial state and try to find a sequence of actions that leads to some state that satisfies the goal condition. Since we work with complete knowledge domains, we demand that the initial state is completely specified.

Definition 10.3.4 Let S be a set of propositional atoms. Then, a set S' of literals on S is complete if for every atom $A \in S$ either $A \in S'$ or $\neg A \in S'$.

Definition 10.3.5 A \mathcal{B} -instance is a triple $\mathcal{I} = \langle \mathcal{D}, S_I, G \rangle$ where $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ is an action description, an initial state S_I and a goal condition G where S_I is a complete consistent set of literals on \mathcal{F} and G is a conjunction of literals from \mathcal{F} .

The Blocks World Example

In the BLOCKS WORLD [210, 98] planning domain the universe consists of a table, a crane, and a set of blocks. The state of the world consists of a description where the blocks lie. They can be either on the table or stacked on top of each other. The only possible operation is that the crane picks up one block and puts it either on the table or on top of another block. Both the block that is moved and the destination block must be the topmost ones of their stacks. It is assumed that the table has enough space to contain all blocks and it will never get full. The effect of the operation is that the block moves to the destination. A plan is a sequence of crane operations that lead to the desired goal state.

Suppose that we have a BLOCKS WORLD instance with n blocks. The action signature for it contains fluents of the form: $on_{i,j}$ and $blocked_i$ where $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, n, \mathbf{t}\}$ where \mathbf{t} denotes the table. The set of action names contains all combinations of $move_{i,j}$ where i and j are defined as before and $i \neq j$. From now on we will use the predicate logic syntax and write $on(i, j)$ instead of $on_{i,j}$ and $move(i, j)$ instead of $move_{i,j}$.

For each block i and place j where $i \neq j$ we have a dynamic law stating that if both i and j are unblocked, we can move i to the top of j :

$$move(i, j) \text{ causes } on(i, j) \text{ if } \neg blocked(i) \wedge \neg blocked(j) .$$

A block that moves leaves its current location:

$$move(i, j) \text{ causes } \neg on(i, k) \text{ if } on(i, k) .$$

This law has the side effect that it prevents us from moving a block that is on the table to another position on the table. Since such moves cannot help us to reach the goal, this is a useful optimization. However, if we wanted to include such moves, we would need to leave out all those laws where $k = \mathbf{t}$.

Finally, we demand that a block that moves has to remain unblocked throughout the move. We need this law for a technical reason to make it possible to create parallel plans. We want to avoid a case where we enter a consistent but nonsensical state where a block i is on the top of j while j is on the top of i . Demanding that a moving block stays unblocked prevents us from creating a cycle of blocks.¹

$$move(i, j) \text{ causes } \neg blocked(i) .$$

A block that has another block on top of it is blocked:

$$blocked(j) \text{ if } on(i, j) .$$

¹We discuss this further in Section 10.5.

We need static laws that ensure that our state stays consistent. First, a block can be in only one place:

$$\perp \text{ if } on(i, j) \wedge on(i, k) \wedge j \neq k .$$

Second, there cannot be more than one block directly on the top of another:

$$\perp \text{ if } on(i, j) \wedge on(k, j) \wedge i \neq k \wedge j \neq t .$$

Finally, we assert that the table is unblocked:

$$\neg blocked(t) .$$

10.3.2 Semantics of \mathcal{B}

The semantics of action languages are defined in terms of labeled transition systems (LTS). In effect, an LTS is a finite automaton where we associate a set of propositional atoms to each state.

Definition 10.3.6 A labeled transition system $\mathbf{L} = \langle \mathbf{S}, \mathbf{T}, \mathbf{R}, \mathbf{P}, \mathbf{V} \rangle$ consists of a finite set of states \mathbf{S} , a set of transition labels \mathbf{T} , a transition relation $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{T} \times \mathbf{S}$, a set of propositional atoms \mathbf{P} , and a valuation $\mathbf{V} : \mathbf{S} \rightarrow 2^{\mathbf{P}}$.

A labeled transition system is *deterministic* if for each state S and a label T , there is at most one transition $\langle S, T, S' \rangle \in \mathbf{R}$. We will consider only deterministic planning domains in this work.

We will assume that the states are complete. That is, an atom p is true in a state S if $p \in \mathbf{V}(S)$ and $\neg p$ is true if $p \notin \mathbf{V}(S)$. The valuation \mathbf{V} generalizes naturally to arbitrary boolean formulas over \mathbf{P} and we use $S \models f$ to denote that the formula f is true in S .

Each action description corresponds to an LTS. Before we define that LTS formally, we need to introduce the concept of logical closure.

Definition 10.3.7 A set S of literals is closed under a set of static laws \mathcal{L}_s if for each law $L \text{ if } F \in \mathcal{L}_s$ it holds that if $S \models F$, then $S \models L$. The set $Cn_{\mathcal{L}_s}(S)$ denotes the smallest consistent set of literals that contains S and is closed under \mathcal{L}_s .

We leave out the subscript from Cn when it is clear from context what the static laws are. Note that we can compute the closure $Cn(S)$ using a modified form of the one-step provability operator from Section 3.4.

When defining the LTS we have to be a bit careful since we defined the valuation to contain only positive atoms while we have both positive and negative fluents in action descriptions. When we have a set of literals S , we use S^+ to denote the positive literals in it.

Definition 10.3.8 An action description $\mathcal{D} = \langle \langle \mathcal{A}, \mathcal{F} \rangle, \mathcal{L}_s, \mathcal{L}_d \rangle$ induces an LTS $\mathbf{L}(\mathcal{D}) = \langle \mathbf{S}, \mathbf{T}, \mathbf{R}, \mathbf{P}, \mathbf{V} \rangle$ where:

1. $\mathbf{P} = \mathcal{F}$;
2. $\mathbf{T} = 2^{\mathcal{A}}$;

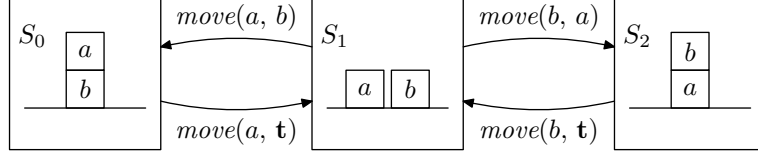


Figure 10.2: The LTS corresponding to a two-block BLOCKS WORLD

3. \mathbf{S} contains all complete consistent sets of literals S over \mathbf{P} that are closed under \mathcal{L}_s ;
4. $\mathbf{V}(S) = S^+$; and
5. \mathbf{R} contains all triples $\langle S, \mathbf{A}, S' \rangle$ such that $S, S' \in \mathbf{S}$, $\mathbf{A} \in \mathbf{T}$ and

$$S' = \text{Cn}_{\mathcal{L}_s}(E(\mathbf{A}, S) \cup (S \cap S')) \quad (10.1)$$

where $E(\mathbf{A}, S)$ is the set:

$$E(\mathbf{A}, S) = \bigcup_{A \in \mathbf{A}} \{L \mid A \text{ causes } L \text{ if } F \in \mathcal{L}_d \text{ and } S \models F\}.$$

An action $\mathbf{A} = \{A_1, \dots, A_n\}$ of the LTS corresponds to taking the elementary actions A_i of the action description in parallel. We usually leave the brackets out when there is only one elementary action in an action.

The intuition behind (10.1) is that there are three types of literals in the new state: *direct* changes that come from the dynamic laws, *indirect* changes that come from the static laws, and literals that keep their values by *inertia*. The new state should be the closure of the *direct effects* $E(\mathbf{A}, S)$ and the set of literals $S \cap S'$ that keep their values under the static laws.

Definition 10.3.9 Given a \mathcal{B} -instance $\mathcal{I} = \langle \mathcal{D}, S_I, G \rangle$ and its induced LTS $\mathbf{L}(\mathcal{I})$, a plan is a sequence of actions $\{\mathbf{A}_1, \dots, \mathbf{A}_n\}$ such that there exists a sequence of states $\langle S_1, \dots, S_{n+1} \rangle$ where

1. $\langle S_i, \mathbf{A}_i, S_{i+1} \rangle \in \mathbf{R}$ for all $i \in [1, n]$;
2. $S_1 = S_I$; and
3. $S_n \models G$.

A \mathcal{B} -instance is satisfiable if and only if it has a plan. A plan is sequential if $|\mathbf{A}_i| \leq 1$ for all $i \in [1, n]$ and parallel, otherwise.

Continuing the Blocks World example

Figure 10.2 shows the LTS corresponding to the BLOCKS WORLD with two blocks. There are three states that we are interested in: S_0 , S_1 , and

S_2 .² The positive parts of the states are :

$$\begin{aligned} \mathbf{V}(S_0) &= \{on(a, b), on(b, \mathbf{t}), blocked(b)\} \\ \mathbf{V}(S_1) &= \{on(a, \mathbf{t}), on(b, \mathbf{t})\} \\ \mathbf{V}(S_2) &= \{on(a, \mathbf{t}), on(b, a), blocked(a)\} . \end{aligned}$$

To see that $move(a, b)$ leads to S_0 from S_1 we need to examine the dynamic and static laws of the action description.

$$E(\{move(a, b)\}, S_1) = \{on(a, b), \neg on(a, \mathbf{t}), \neg blocked(a)\} .$$

The intersection of the two states is:

$$S_0 \cap S_1 = \{on(b, \mathbf{t}), \neg blocked(\mathbf{t})\}$$

Thus, we need to find the closure of:

$$\begin{aligned} S' &= E(\{move(a, b)\}, S_1) \cup (S_0 \cap S_1) \\ &= \{on(a, b), \neg on(a, \mathbf{t}), \neg blocked(a), on(b, \mathbf{t}), \neg blocked(\mathbf{t})\} . \end{aligned}$$

The static law $blocked(b)$ **if** $on(a, b)$ forces us to add $blocked(b)$ to the set, but no other static law has a satisfied body. Thus, the positive part of the closure is:

$$Cn(S') = \{on(a, b), on(b, \mathbf{t}), blocked(b)\} = S_1 .$$

The other transitions work in the same way. The action $move(a, \mathbf{t})$ cannot be taken in the state S_1 since

$$\{on(a, \mathbf{t}), \neg on(a, \mathbf{t})\} \in E(move(a, \mathbf{t}), S_1)$$

so the set of effects is not consistent.

Note that in this small example there are no parallel transitions. Trying to apply $A = \{move(a, b), move(b, a)\}$ leads into an inconsistency since

$$\begin{aligned} \neg blocked(a) &\in E(move(a, b), S_1) \text{ and} \\ blocked(a) &\in Cn(E(move(b, a), S_1)) . \end{aligned}$$

10.3.3 On Preconditions and Effects

In STRIPS [73] based planning formalisms there are two kinds of conditions for actions:

1. A *precondition* that has to be satisfied before we can execute an action; and
2. A *condition for effect* that controls whether we apply some particular effect of an executed action.

²The positive part of the fourth closed set of literals is $\{on(a, b), on(b, a), blocked(a), blocked(b)\}$ that is nonsensical. We leave it out since it is not reachable from any other state.

We have only conditional effects in the formal semantics of \mathcal{B} . When we take an elementary action A , we look through all of its dynamic laws A **causes** L **if** F and add L to the effects if F is true in the current state.

This approach has one conceptual problem. When all dynamic laws for A have unsatisfied conditions F , then the set of active effects $E(A, S)$ is empty. Then, for each action $\mathbf{A} = \{A_1, \dots, A_n\}$ it holds that $E(\mathbf{A}, S) = E(\mathbf{A} \cup \{A\}, S)$ so we can arbitrarily choose to either include A or not in the plan without affecting its validity.

For example, suppose that we have an elementary action $move(x, a, b)$ that moves an object x from a to b and in our current state x is not at a . The set of its active effects is empty and by the semantics we can choose to include it in the action at that step. Then, when we have the plan we need to filter the empty actions out from it so that we do not try to actually move x from a when it is not there.

We will use the convention that an action cannot contain any elementary actions with an empty set of active effects. We call them *no-op* actions.

Definition 10.3.10 *An elementary action A is no-op in a state S if and only if $E(\{A\}, S) = \emptyset$. A plan is proper if no action in it contains a no-op elementary action.*

It is easy to see that if a \mathcal{B} -instance is satisfiable, then it has a proper plan.

Lemma 10.3.1 *If a \mathcal{B} -instance has a plan, then it has a proper plan.*

Proof. Let $p = \langle \mathbf{A}_1, \dots, \mathbf{A}_n \rangle$ be a minimum length plan of the instance and $\langle S_1, \dots, S_{n+1} \rangle$ the corresponding sequence of states. Consider the sequence $p' = \langle \mathbf{A}'_1, \dots, \mathbf{A}'_n \rangle$ where

$$\mathbf{A}'_i = \{A \mid A \in \mathbf{A}_i \text{ and } E(A, S_i) \neq \emptyset\} .$$

Now $E(\mathbf{A}'_i, S_i) = E(\mathbf{A}_i, S_i)$, so p' is also a plan. By construction p' is proper if $\mathbf{A}'_i \neq \emptyset$ for all i . Suppose that this is not the case. Then, $E(\mathbf{A}_i, S_i) = \emptyset$ so $S_{i+1} = S_i$. This means that $\langle \mathbf{A}_1, \dots, \mathbf{A}_{i-1}, \mathbf{A}_{i+1}, \dots, \mathbf{A}_n \rangle$ is a plan which contradicts our assumption that p has a minimum length. \square

It is often more convenient to formalize a planning domain if we have explicit preconditions. We can simulate them using the conditional effects by adding the precondition into the condition of every dynamic law of an action. For example, the precondition for the BLOCKS WORLD *move* action is that both blocks should be unblocked, so we include that in all three dynamic laws for *move*:

$$\begin{aligned} move(i, j) & \textbf{causes } on(i, j) \textbf{ if } \neg blocked(i) \wedge \neg blocked(j) \\ move(i, j) & \textbf{causes } \neg on(i, k) \textbf{ if } on(i, k) \wedge \neg blocked(i) \wedge \neg blocked(j) \\ move(i, j) & \textbf{causes } \neg blocked(i) \textbf{ if } \neg blocked(i) \wedge \neg blocked(j) . \end{aligned}$$

We will use the convention that we interpret the common part of the conditions of an action as the precondition and the other parts as conditional effects.

Definition 10.3.11 Let $\mathcal{D} = \langle \langle \mathcal{A}, \mathcal{F} \rangle, \mathcal{L}_s, \mathcal{L}_d \rangle$ be an action description, $A \in \mathcal{A}$ be an elementary action in it, and \mathcal{L}_A the set of dynamic laws for A . Then, the precondition P_A of A is the set:

$$P_A = \bigcap_{\mathcal{L} \in \mathcal{L}_A} \text{cond}(\mathcal{L})$$

and the set of conditional effects E_A of A is the set of pairs:

$$E_A = \{ \langle L, C_1 \wedge \dots \wedge C_n \rangle \mid \\ A \text{ causes } L \text{ if } C_1 \wedge \dots \wedge C_n \wedge F_1 \wedge \dots \wedge F_m \in \mathcal{L}_A \\ \text{and } P_A = \{F_1, \dots, F_m\} \} .$$

10.3.4 Computational Complexity of Planning

Different forms of planning belong to different complexity levels. We will be examining the two following problems.

PROBLEM 10: PLANNING. Does a \mathcal{B} -instance $\langle \mathcal{D}, S_I, G \rangle$ have a plan?

PROBLEM 11: BOUNDED PLANNING. Does a \mathcal{B} -instance $\langle \mathcal{D}, S_I, G \rangle$ have a plan that is at most t steps long?

The general PLANNING is **PSPACE**-complete for the STRIPS planning definition language in the propositional case [18]. Since \mathcal{B} can express STRIPS descriptions, we know that it is **PSPACE**-hard for \mathcal{B} -planning.

Turner [201] analyzed an action-based formalism and showed that BOUNDED PLANNING is **NP**-complete for polynomial-length plans when the domain has complete knowledge and deterministic actions. The \mathcal{B} action descriptions can be translated to Turner's formalism in polynomial time and vice versa, so BOUNDED PLANNING is also **NP**-complete for the same domains.

We will define our encodings for BOUNDED PLANNING. We then solve PLANNING by using BOUNDED PLANNING as an oracle that is called with different values for the time bound t .

10.4 TRANSLATING ACTION LANGUAGE \mathcal{B} TO ASP

In this section we show how we can translate \mathcal{B} to cardinality constraint programs. We take as an input a \mathcal{B} -instance together with an integer t and construct a program whose answer sets correspond to the plans that are at most t steps long.

In the translation we have to choose how to represent fluents and actions as well as how to implement the changes in states when we take an action.

Translating Action Languages

1. Every fluent F is represented with t atoms $fluent_F(i)$ where i encodes the time information.
2. Every action A is represented with t atoms $action_A(i)$. An atom $action_A(i)$ is true when A is taken on the i th time step.
3. All dynamic laws A **causes** L **if** F for an action A are considered together. The literals F_i that occur in the conditions of all laws become the precondition of A . The rest of the literals F' become specific conditions for the particular effects L .

The rules for an action have the form:

$$\begin{aligned} \{action_A(i)\} &\leftarrow precondition(i) \\ fluent_L(i+1) &\leftarrow action(i), condition(i) . \end{aligned}$$

4. A static law L **if** F is encoded with rules:

$$fluent_L(i) \leftarrow fluent_F(i) .$$

5. The inertia of a fluent F is defined with rules of the form:

$$\begin{aligned} fluent_F(i+1) &\leftarrow fluent_F(i), \text{not } \neg fluent_F(i+1) \\ \neg fluent_F(i+1) &\leftarrow \neg fluent_F(i), \text{not } fluent_F(i+1) . \end{aligned}$$

6. The initial state $S_I = \{F_1, \dots, F_n\}$ is encoded as:

$$\begin{aligned} fluent_{F_1}(1) &\leftarrow \\ &\vdots \\ fluent_{F_n}(1) &\leftarrow . \end{aligned}$$

7. The goal condition $G = \{F_1, \dots, F_n\}$ is encoded with:

$$\begin{aligned} &\leftarrow \text{not } fluent_{F_1}(t+1) \\ &\vdots \\ &\leftarrow \text{not } fluent_{F_n}(t+1) \end{aligned}$$

Figure 10.3: The action language translation in a nutshell

10.4.1 Fluents

We have to be able to tell when a fluent is true or false so we add a new argument to the predicate symbols to encode the time information.

Definition 10.4.1 *Let \mathcal{F} be a set of fluents and i an integer. Then, the planning translation $\text{tl}(F, i)$ of $F \in \mathcal{F}$ is the atom $\text{fluent}_F(i)$, and the translation of a literal $\neg F$ is the literal $\neg \text{fluent}_F(i)$. If S is a set of literals on \mathcal{F} , then*

$$\text{tl}(S, i) = \{\text{tl}(L, i) \mid L \in S\} .$$

If $F_\wedge = F_1 \wedge \dots \wedge F_n$ is a conjunction of literals on \mathcal{F} , then

$$\text{tl}(F_\wedge, i) = \{\text{tl}(F_k, i) \mid k \in [1, n]\} .$$

The intuition is that the argument i tells when exactly the fluent is true.

In the examples we will be using a syntax that is more readable. As we mentioned before, a fluent $on_{x,y}$ is customarily written as $on(x, y)$. We will augment this custom by adding the time parameter directly to the argument list of on so we use $on(x, y, i)$ instead of $\text{fluent}_{on_{x,y}}(i)$.

Inertia

According to our semantics, a fluent keeps its value if an action does not change it. This property is called *inertia*. We can encode this property with rules of the form:

$$\begin{aligned} \text{fluent}(i+1) &\leftarrow \text{fluent}(i), \text{not } \neg \text{fluent}(i+1) \\ \neg \text{fluent}(i+1) &\leftarrow \neg \text{fluent}(i), \text{not } \text{fluent}(i+1) \end{aligned}$$

If $\text{fluent}(i)$ is true, then we set $\text{fluent}(i+1)$ true unless some action that we take has $\neg \text{fluent}(i+1)$ among its direct effects or indirect consequences.

Definition 10.4.2 *Let \mathcal{F} be a set of fluents and t an integer. Then, the inertia translation $\text{inertia}(F, t)$ of a fluent $F \in \mathcal{F}$ is the set of rules:*

$$\begin{aligned} \text{inertia}(F, t) = \{ &\langle \text{tl}(F, i+1), \{\text{tl}(F, i), \text{not } \neg \text{tl}(F, i+1)\} \rangle \mid 1 \leq i \leq t \} \cup \\ &\{ \langle \neg \text{tl}(F, i+1), \{\neg \text{tl}(F, i), \text{not } \text{tl}(F, i+1)\} \rangle \mid 1 \leq i \leq t \} \end{aligned}$$

The inertia translation $\text{inertia}(\mathcal{F}, t)$ is the set

$$\text{inertia}(\mathcal{F}, t) = \bigcup_{F \in \mathcal{F}} \text{inertia}(F, t) .$$

Strong Negation

When we use strong negation in a CCP, we can have the case where neither L nor $\neg L$ is true in an answer set. In this planning formalization we are working with completely defined states, so we add rules that force $\neg L$ to be true if L is false. However, we can encode many if not most planning domains without having to use strong negations. In those cases we can leave the totalizing rules out of the encoding.

Definition 10.4.3 Let \mathcal{F} be a set of fluents and t an integer. Then, the totalizing translation $\text{total}(F, t)$ of a fluent $F \in \mathcal{F}$ is the set of rules:

$$\text{total}(F, t) = \{ \langle \neg \text{tl}(F, i), \{ \text{not tl}(F, I) \} \rangle \mid 1 \leq i \leq t \}$$

The translation $\text{total}(\mathcal{F}, t)$ is the set:

$$\text{total}(\mathcal{F}, t) = \bigcup_{F \in \mathcal{F}} \text{total}(F, t) .$$

10.4.2 Actions

We add the time information to the actions in exactly the same way as we do with the fluents. Formally, we will have atoms of the form $\text{action}_A(i)$, but in practice we will use the elementary action as the predicate symbol and add the time information as the final argument. For example, an action $\text{move}_{a,b}$ is represented with $\text{move}(a, b, i)$ instead of $\text{action}_{\text{move}_{a,b}}(i)$.

We use the following general form to encode actions:

$$\begin{aligned} \{ \text{action}(i) \} &\leftarrow \text{preconditions}(i) \\ \text{effect}(i+1) &\leftarrow \text{action}(i), \text{condition}(i) . \end{aligned}$$

The intuition is that if the preconditions hold at the time step i , then we can choose to take an action, but we do not have to do so. If we choose to take the action, then all those effects whose conditions are currently satisfied will be true at step $i+1$.

Definition 10.4.4 Let \mathcal{A} be a set of elementary actions and i be an integer. Then, the planning translation $\text{tl}(A, i)$ of $A \in \mathcal{A}$ is the atom $\text{action}_A(i)$ and $\text{tl}(\mathcal{A}, i) = \{ \text{tl}(A, i) \mid A \in \mathcal{A} \}$.

Definition 10.4.5 Let $\mathcal{D} = \langle \langle \mathcal{A}, \mathcal{F} \rangle, \mathcal{L}_s, \mathcal{L}_d \rangle$ be an action description and t be an integer.

Then, the action translation $\text{act}(A, t)$ of an elementary action $A \in \mathcal{A}$ is the set of rules:

$$\begin{aligned} \text{act}(A, t) = & \{ \langle \{ \text{tl}(A, i) \}, \text{tl}(P_A, i) \rangle \mid 1 \leq i \leq t \} \cup \\ & \{ \langle \text{tl}(L, i+1), \{ \text{tl}(A, i) \} \cup \text{tl}(C, i) \rangle \mid \langle L, C \rangle \in E_A, 1 \leq i \leq t \} . \end{aligned}$$

The planning translation $\text{acts}(\mathcal{L}_d, t)$ is the set:

$$\text{acts}(\mathcal{L}_d, t) = \bigcup_{A \in \mathcal{A}} \text{act}(A, t) .$$

Example 10.4.1 Consider the $\text{move}(a, b)$ action from BLOCKS WORLD when we have only two blocks. The translation $\text{act}(\text{move}(a, b), t)$ is the set of rules:

$$\begin{aligned} \{ \text{move}(a, b, i) \} &\leftarrow \neg \text{blocked}(a, i), \neg \text{blocked}(b, i) \\ \text{on}(a, b, i+1) &\leftarrow \text{move}(a, b, i) \\ \neg \text{on}(a, \text{table}, i+1) &\leftarrow \text{move}(a, b, i), \text{on}(a, \text{table}, i) \\ \neg \text{blocked}(a, i+1) &\leftarrow \text{move}(a, b, i) \end{aligned}$$

where i ranges from 1 to t .

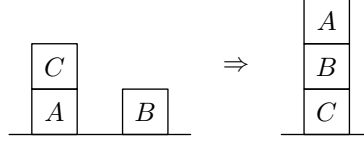


Figure 10.4: The “Sussman Anomaly” BLOCKS WORLD instance

10.4.3 Static Laws

Static laws are straightforward to encode, since they can be directly mapped into basic rules. A law L **if** F becomes:

$$fluent_L(i) \leftarrow fluent_F(i)$$

where i ranges over all time steps.

Definition 10.4.6 Let \mathcal{L}_s be a set of static laws and t an integer . Then, the planning translation $law(S, t)$ of a static law $S = L$ **if** $F_\wedge \in \mathcal{L}_s$ is:

$$law(S, t) = \{ \langle tl(L, i), tl(F_\wedge, i) \rangle \mid 1 \leq i \leq t \} .$$

The translation $laws(\mathcal{L}_s, t)$ is the set:

$$laws(\mathcal{L}_s, t) = \bigcup_{S \in \mathcal{L}_s} law(S, t) .$$

When we translate the static laws for a t step planning instance, we want to create the rules up to the time step $t + 1$. This ensures that the world state will be consistent also after we take the last action.

10.4.4 Putting Everything Together

When we have a \mathcal{B} encoding, we can create a corresponding ground CCP by putting together all previous translations.

Definition 10.4.7 Let $\mathcal{D} = \langle \langle \mathcal{A}, \mathcal{F} \rangle, \mathcal{L}_s, \mathcal{L}_d \rangle$ be an action description and t an integer. Then, $tl(\mathcal{D}, t)$ is the ground CCP:

$$tl(\mathcal{D}, t) = acts(\mathcal{L}_d, t) \cup laws(\mathcal{L}_s, t + 1) \cup inertia(\mathcal{F}, t) \cup total(\mathcal{F}, t + 1) .$$

When we have a planning instance, we add the initial and goal states to the translation of the domain. We make all fluents that belong to the initial state true at the time step 1 and we add constraints to ensure that all fluents in the final state are true at the time step $t + 1$.

Definition 10.4.8 Let $\mathcal{I} = \langle \mathcal{D}, S_I, G \rangle$ be a \mathcal{B} -instance and t is an integer. Then, $tl(\mathcal{D}, S_I, G, t)$ is the ground CCP:

$$tl(\mathcal{D}, S_I, G, t) = tl(\mathcal{D}, t) \cup \{ \langle tl(F, 1), \emptyset \rangle \mid F \in S_I \} \cup \{ \langle \perp, \{ \text{not } tl(F, t + 1) \} \rangle \mid F \in G \} .$$

Example 10.4.2 Consider the BLOCKS WORLD instance from Figure 10.4.³. The translation of the initial state is:

$$\begin{aligned} \text{on}(c, a, 1) &\leftarrow \\ \text{on}(a, \text{table}, 1) &\leftarrow \\ \text{on}(b, \text{table}, 1) &\leftarrow \end{aligned}$$

When we want to find a three-step plan, the translation of the goal condition becomes:

$$\begin{aligned} &\leftarrow \text{not } \text{on}(a, b, 4) \\ &\leftarrow \text{not } \text{on}(b, c, 4) \\ &\leftarrow \text{not } \text{on}(c, \text{table}, 4) . \end{aligned}$$

10.4.5 Correctness of the Translation

Next, we show that our translation from \mathcal{B} to logic programs is correct. We do it in two steps. First we prove that if a \mathcal{B} -instance $\langle \mathcal{D}, S_I, G \rangle$ has a plan, then the translation $\text{tl}(\mathcal{D}, S_I, G, t)$ has a stable model for some t . Next, we show that if the translation has a stable model, then there exists a corresponding plan.

Theorem 10.4.1 *If a \mathcal{B} -instance $\mathcal{I} = \langle \mathcal{D}, S_I, G \rangle$ is satisfiable, then there exists $t \in \mathbb{N}$ such that the translation $\text{tl}(\mathcal{D}, S_I, G, t)$ has a stable model.*

Proof. By Lemma 10.3.1 \mathcal{I} has a proper plan $\{\mathbf{A}_1, \dots, \mathbf{A}_n\}$ and an associated sequence of states $\langle S_1, \dots, S_{n+1} \rangle$ where $S_I = S_1$ and $S_{n+1} \models G$.

We now argue that $\text{tl}(\mathcal{D}, S_I, G, n)$ has a stable model. In particular, the model is:

$$M = \{\text{tl}(A, k) \mid A \in \mathbf{A}_k, 1 \leq k \leq n\} \cup \{\text{tl}(L, k) \mid L \in S_k, 1 \leq k \leq n+1\} .$$

We start by noting that since all states S_k satisfy the static laws of the action description, M satisfies all constraints that belong to $\text{laws}(\mathcal{D})$.

Next, we define $t + 1$ sets U_k that contain all fluents that belong to time step k or earlier and all actions up to $k - 1$.

$$\begin{aligned} U_k &= \{\text{tl}(F, i) \mid F \in \mathcal{F} \text{ and } 1 \leq i \leq k\} \\ &\quad \cup \{\text{tl}(A, i) \mid A \in \mathcal{A} \text{ and } 1 \leq i \leq k - 1\} . \end{aligned}$$

As no atom in P depends on atoms that occur in later time steps, each set U_k is a splitting set on P . Moreover, when M' is a stable model of the bottom program B_{U_k} , the set U_{k+1} is a splitting set for the modified top program $P(U_k, M') = T_{U_k} \cup F(M')$.

Thus, by Corollary 5.1.1 we can split any stable model M' of P into a sequence of $t + 1$ sets $\langle M_1, \dots, M_{t+1} \rangle$ where $M_k \subset M_{k+1}$ for all $k \leq t$, $M' = M_{t+1}$, and each M_k is a stable model of B_{U_k} . We now show by induction that we can find such a sequence for M .

³This example is called the Sussman Anomaly because Gerald Sussman noticed that the original STRIPS planner could not solve it at all [185].

In the base case $k = 1$ we have the set $M_1 = \{\text{tl}(L, 1) \mid L \in S_1\}$. The fluents that belong to the initial state are given as the set of facts $\mathcal{F}_1 = \{\langle \text{tl}(F, 1), \emptyset \rangle \mid F \in S_I\}$. Since S_I is consistent and complete, $\mathcal{F}_1 = M_1$.

Next, suppose that the claim holds up to some $n < t$ and consider $k = n + 1$. The set M_{n+1} can be divided into two distinct sets $M_{n+1} = M_n \cup M'_{n+1}$. Since U_n is a splitting set also for $B_{U_{n+1}}$ and by induction hypothesis M_n is a stable model of B_{U_n} , so we have to consider only the rules in $B = B_{U_{n+1}} \setminus B_{U_n}$.

Since the plan is proper and we could execute \mathbf{A}_n , we know that S_n satisfies the precondition P_A for each action $A \in \mathbf{A}_n$. Thus, the body of the rule:

$$\{\text{tl}(A, n)\} \leftarrow \text{tl}(P_A, n)$$

is satisfied and we have a justification to include $\text{tl}(A, n)$ in the stable model.

For each effect $\langle L, C \rangle \in E_A$ we have the rule:

$$\text{tl}(L, n + 1) \leftarrow \text{tl}(A, n), \text{tl}(C, n) .$$

If $S_n \models C$, then $\text{tl}(C, n) \in M_n$ by induction hypothesis and this rule derives the atom $\text{tl}(L, n + 1) \in M_{n+1}$. On the other hand, if $S_n \not\models C$, $\text{tl}(C, n) \notin M_n$ so this rule cannot incorrectly add the head into M_{n+1} .

The program P has the following inertia rules for each fluent $\text{tl}(F, n + 1)$

$$\begin{aligned} \text{tl}(F, n + 1) &\leftarrow \text{tl}(F, n), \text{not } \neg \text{tl}(F, n + 1) \\ \neg \text{tl}(F, n) &\leftarrow \neg \text{tl}(F, n), \text{not } \neg \text{tl}(F, n + 1) . \end{aligned}$$

If $F \in S_n$, $\text{tl}(F, n) \in M_n$. Unless we derived $\neg \text{tl}(F, n)$ because $\langle \neg F, C \rangle \in E_A$ for some action $A \in \mathbf{A}_n$, we can use the first rule to derive $\text{tl}(F, n + 1) \in M_{n+1}$. The second rule ensures the same for $\neg F$. Thus, M_{n+1} is a stable model of $B_{U_{n+1}}$ and M is a stable model of $\text{tl}(\mathcal{D}, S_I, G, n)$. \square

Theorem 10.4.2 *If $\text{tl}(\mathcal{D}, S_I, G, t)$ has a stable model, then $\mathcal{I} = \langle \mathcal{D}, S_I, G \rangle$ is a satisfiable \mathcal{B} -instance.*

Proof. Let M be a stable model of $P = \text{tl}(\mathcal{D}, S_0, G, t)$. We define the set \mathbf{A}_k for all $1 \leq k \leq t$:

$$\mathbf{A}_k = \{A \mid \text{tl}(A, k) \in M\} .$$

Next, we argue that the sequence

$$p = \langle \mathbf{A}_1, \dots, \mathbf{A}_t \rangle$$

is a plan for \mathcal{I} .

Consider the sets:

$$\begin{aligned} M_k &= \{\text{tl}(L, k) \mid L \text{ is a fluent literal and } 1 \leq k \leq t + 1\} \\ S_k &= \{L \mid \text{tl}(L, k) \in M_k\} . \end{aligned}$$

We have to show that:

1. all sets S_k are states in $\mathbf{L}(\mathcal{I})$;
2. $\langle S_k, \mathbf{A}_k, S_{k+1} \rangle \in \mathbf{R}$;
3. $S_1 = S_I$; and
4. $S_{t+1} \models G$.

Conditions 3. and 4. follow directly from the construction of P .

Next, we prove that if S_k and S_{k+1} are states, then $\langle S_k, \mathbf{A}_k, S_{k+1} \rangle$ is a transition in the induced LTS $\mathbf{L}(\mathcal{I})$.

First we consider the base case $k = 1$. By Theorem 3.6.2 every atom in a stable model has to have a justification. Since the only way to derive an atom $\text{tl}(A, 1)$ is with a rule of the form:

$$\{\text{tl}(A, 1)\} \leftarrow \text{tl}(P_A, 1)$$

where P_A is the precondition of A , we know that $S_1 \models P_A$. Consider a conditional effect $\langle L, C \rangle \in E_A$. If $S_1 \models C$, then the body of:

$$\text{tl}(L, 2) \leftarrow \text{tl}(A, 1), \text{tl}(C, 1) \tag{10.2}$$

is true so $\text{tl}(L, 2) \in M_2$. Thus, $E(\mathbf{A}_1, S_1) \subseteq S_2$.

Next, consider a static law

$$L \text{ if } F$$

If $M_2 \models \text{tl}(F, 2)$, then the body of the rule:

$$\text{tl}(L, 2) \leftarrow \text{tl}(F, 2) \tag{10.3}$$

is true and $\text{tl}(L, 2) \in M_2$. Thus, S_2 is closed under the static laws.

If M_2 contains a literal L that was not derived by rules of the form (10.2) or (10.3), then it was added there by a rule in $\text{inertia}(\mathcal{F}, 2)$ of the form:

$$\text{tl}(L, 2) \leftarrow \text{tl}(L, 1), \text{not } \text{tl}(\bar{L}, 2) . \tag{10.4}$$

Thus, $L \in S_1 \cap S_2$. Putting this together we see that:

$$S_2 = \text{Cn}(E(\mathbf{A}_1, S_1) \cup (S_1 \cap S_2))$$

so $\langle S_1, \mathbf{A}_1, S_2 \rangle \in \mathbf{R}$ provided that S_2 is a valid state of $\mathbf{L}(\mathcal{I})$.

Suppose that there is some n such that for each $k \leq n$ it holds that $\langle S_k, \mathbf{A}_k, S_{k+1} \rangle \in \mathbf{R}$ whenever S_k and S_{k+1} are states.

When $k = n + 1$, we can use exactly the same argument that we used for the base case to show that $\langle S_{k+1}, \mathbf{A}_{k+1}, S_{k+2} \rangle \in \mathbf{R}$.

We now conclude the proof by arguing that each S_k is a state. Since M is a stable model, M_k and thus S_k is consistent. We already proved that S_k is closed under the static laws. It is also complete because S_1 is required to be complete and the rules for inertia guarantee that every literal has a truth value in each following step.

□

10.4.6 Related Work

In this section we compare our translation to other systematic translations from action languages to computational logic formalisms that have been presented in literature [136, 49, 23, 56].

Causal Theories

McCain and Turner [136] presented the first systematic action language translation where they encoded causal theories into propositional logic. A *causal theory* [135] consists of rules of the form:

$$F \Rightarrow G$$

where F and G are propositional formulas. The intended meaning is that if F is true, then there is a cause for G to be true. If a theory is finite and every head G of a rule in it is a literal, then it is *definite*. Intuitively, an interpretation of a theory is *causally explained* if every literal that is true in it is caused by some rule whose body is true in the interpretation.⁴

McCain and Turner [136] present a translation from definite causal theories into propositional satisfiability where they use an analogue of Clark's completion [32] to handle explanations.

For each atom A in a theory the translation includes n atoms $A(i)$ where i contains the time information. A causal law then becomes an implication:

$$F(i) \rightarrow G(i+1) .$$

Since every true literal has to have an explanation, there has to be a formula:

$$G(i+1) \leftrightarrow (F_1(i) \vee \dots \vee F_n(i)) \quad (10.5)$$

where F_1, \dots, F_n are the bodies of the rules that contain G in the head. Intuitively, this formula says that if G is true at a time step $i+1$, then some of its causes has to be true at the time step i .

A definite causal rule $F \Rightarrow G$ can be expressed as a dynamic law:

$$A_F \text{ causes } G \text{ if } F$$

where A_F is a new action name. Thus, definite causal theories correspond to action descriptions that contain only dynamic laws and we can represent them using our translation. Since stable models are justified, we do not need an equivalent of (10.5) as $G(i+1)$ may be true only if it occurs in the head of a rule with a satisfied body.

Normal Logic Programs

Our translation was originally inspired by the one that Lifschitz [113] presented for normal logic programs. In it a dynamic law :

$$A \text{ causes } L \text{ if } F$$

is translated into:

$$L(T_2) \leftarrow A(T_1), F(T_1), \text{next}(T_1, T_2)$$

⁴For the complete formal semantics, see [135].

where $next/2$ encodes the successor relation among time steps. This rule has essentially the same form as our corresponding rule. Similarly, the static laws are encoded in a similar way.

The translation includes generator rules that allow us to choose the truth values of both the fluents and actions freely by including rules of the form:

$$\begin{aligned} F(T) &\leftarrow \text{not } F'(T) \\ F'(T) &\leftarrow \text{not } F(T) \\ A(T) &\leftarrow \text{not } A'(T) \\ A'(T) &\leftarrow \text{not } A(T) \end{aligned}$$

for each fluent F and action A . Here F' is an atom that denotes the strong negation $\neg F$. These rules make it possible to always execute an action unless some other rule explicitly forbids it and they make every fluent non-inertial unless otherwise specified.

The rules for inertia have the form:

$$\begin{aligned} F(T_2) &\leftarrow F(T_1), \text{not } F'(T_2), next(T_1, T_2) \\ F'(T_2) &\leftarrow F'(T_1), \text{not } F(T_2), next(T_1, T_2) . \end{aligned}$$

Again, these rules are essentially the same as ours.

There are two main differences between our tl and the normal logic program encoding:

1. we add an explicit precondition to control when we can include an atom $tl(A, i)$ in an answer set; and
2. we do not have rules that allow an arbitrary truth value choice for fluents by default.

Adding explicit preconditions removes no-op actions from the plans making them easier to interpret. Also, having an arbitrary truth value choice can increase the effective search space size since they are defined also for inertial fluents that do not need them.

Doğandag et. al. [49, 50] presented a modified version of Lifschitz's translation. They added a direct support for different fluent types.⁵

The translation of a fluent F is an atom $holds(F, I)$ where I is the time information, and the translation of an action A is the atom $occurs(A, I)$.

Then, a dynamic law:

$$A \text{ causes } L \text{ if } F$$

is translated as:

$$holds(L, I + 1) \leftarrow occurs(A, I), holds(F, I) .$$

This is essentially the same as Lifschitz's or our corresponding rule. The encoding has also generator rules that give a free choice of both fluents and actions.

⁵We will examine these fluent variants in Section 10.8.1.

Different properties of fluents are specified with additional predicate symbols. For example, $inertialFluent(F)$ denotes that F is inertial, and $defaultTrue(F)$ states that F should default to true unless some other law causes it to be false. The rules for inertia are:

$$\begin{aligned} holds(F, I + 1) &\leftarrow holds(F, I), \text{not } \neg holds(F, I + 1), inertialFluent(F) \\ \neg holds(F, I + 1) &\leftarrow \neg holds(F, I), \text{not } holds(F, I + 1), inertialFluent(F) . \end{aligned}$$

The only real difference between this and the other inertial rules is it has an explicit atom $inertialFluent(F)$ in the body to control when it is applied.

Using only one predicate symbol for all fluents and one for all actions has the advantage that it makes it easy to define properties for groups of fluents or actions. The downside is that it makes it more difficult to use domain-specific knowledge to optimize the encoding of some fluent.

Disjunctive Logic Programs

In a disjunctive logic program [60] rules have the form:

$$A_1 \vee \dots \vee A_n \leftarrow body$$

where A_i are literals. The semantics is such that I is an answer set if and only if it is a minimal model of the reduct P^I that is taken in the same way as in normal logic programs (See Section 3.1).

Eiter et. al. present a translation [56] from the action language \mathcal{K} to disjunctive logic programs. The language \mathcal{K} is designed to support incomplete knowledge and default negation.

Our translation is very similar to the one presented in [56]. The main difference is that we extract the common part of the conditions of dynamic laws into a precondition.

The \mathcal{K} -language supports explicit preconditions in the form of **executable** declarations. A statement:

$$\text{executable } A \text{ if } P$$

is translated into the disjunctive rule:

$$A(I) \vee \neg A(I) \leftarrow P(I)$$

where I is again the time information. A rule of this form imposes a choice of including either $A(I)$ or $\neg A(I)$ into the program if $P(I)$ is true.

Within the scope of the \mathcal{B} language, the handling of choices and preconditions are the only noticeable differences between our translation and the one in [56]. However, later we will see a couple of examples of cases where we use cardinality atoms to get concise translations of some action language features.

10.5 ISSUES ON PARALLEL PLANNING

There are two different kinds of parallelism in action descriptions. An action $\mathbf{A} = \{A_1, A_2\}$ can either represent a situation where two different

actors take A_1 and A_2 concurrently, or a situation where a single actor executes them in some order.

For example, in BLOCKS WORLD we have only one crane so if take the action $\mathbf{A} = \{\text{move}(i, j), \text{move}(m, n)\}$, we actually execute the elementary actions in some sequential order. This kind of a representation where we allow an actor to take more than one independent action at the same step is essential to computational efficiency [106]. The more time steps we have in a plan, the larger our search space is, so cutting down the plan length reduces the effort needed for finding an answer. Intuitively, we are at the same time examining all sequential plans that contain the actions from \mathbf{A} in some order.

The problem that we face is that a parallel plan $\langle \mathbf{A}_1, \dots, \mathbf{A}_n \rangle$ might not correspond to any sequential plan at all.

Example 10.5.1 *Consider the action description:*

$$\begin{aligned} A_1 &\text{ causes } F_1 \text{ if } \neg F_2 \\ A_2 &\text{ causes } F_2 \text{ if } \neg F_1 \end{aligned}$$

Both elementary actions are executable in the state $S = \{\neg F_1, \neg F_2\}$, and $E(\{A_1, A_2\}, S) = \{F_1, F_2\}$ that is a consistent set satisfying all static laws. However, if we first take A_1 , we end up in a state $\{F_1, \neg F_2\}$ and A_2 is not executable there. Similarly, taking A_2 first leads to $\{\neg F_1, F_2\}$ and we cannot execute A_1 .

In the rest of this section we examine how we can ensure that a parallel plan $p = \langle \mathbf{A}_1, \dots, \mathbf{A}_n \rangle$ corresponds to some linear plan.

Definition 10.5.1 *Let \mathcal{I} be an action instance and $\langle \mathbf{S}, \mathbf{T}, \mathbf{R}, \mathbf{P}, \mathbf{V} \rangle$ be its induced LTS. A linearization of an action $\mathbf{A} = \{A_1, \dots, A_n\}$ in a state $S_1 \in \mathbf{S}$ is a sequence of actions $\langle A'_1, \dots, A'_n \rangle$ such that there exists a sequence of states $\langle S_1, \dots, S_{n+1} \rangle$ where $\langle S_i, \{A_i\}, S_{i+1} \rangle \in \mathbf{R}$ for all $i \in [1, n]$ and $\langle S_1, \mathbf{A}, S_{n+1} \rangle$.*

A linearization of a plan $\langle \mathbf{A}_1, \dots, \mathbf{A}_m \rangle$ is a sequence of actions

$$\langle A'_{11}, \dots, A'_{1n_1}, \dots, A'_{m1}, \dots, A'_{mn_m} \rangle$$

where each subsequence $\langle A'_{i1}, \dots, A'_{in_i} \rangle$ is a linearization of \mathbf{A}_i .

PROBLEM 12: LINEARIZATION. Does a plan of a \mathcal{B} -instance have a linearization?

There have been two general approaches for the linearization problem: *universal* [28, 106] and *existential* [47, 166] step plans. Under universal step semantics we demand that every linearization of a plan has to be a correct sequential plan while under existential step semantics it is enough if at least one linearization is a correct plan. In this section we consider mostly universal step plans.

Our basic approach is that we try to handle the possible problems at the level of action descriptions: when we need plans that can be linearized, we create a description that guarantees it.

10.5.1 Possible Problem Sources

If we want to create an action description for a sequential domain that allows parallel planning, we have to consider several things that can prevent linearization of a parallel plan:

1. two actions may have contradictory effects;
2. one of the actions may invalidate a precondition of another; and
3. applying actions in parallel may lead us into a state that we could not reach with any linearization of the compound action.

Strictly speaking 2. could be seen as a special case of 3 but we treat it separately since it can occur in practically all planning domains, even in those that are inherently parallel.

We will next examine these three problem sources.

10.5.2 Contradictions

The semantics of \mathcal{B} automatically handles cases where two elementary actions A_1 and A_2 have F and $\neg F$ among their effects. States have to be consistent, so trying to take both actions at the same time fails.

A situation is more complex when two different fluents are used to model one object in the planning. For example, if we can move an object O from place A to place B , we have the fluents $at(o, a)$ and $at(o, b)$. A parallel planner does not know how we interpret the fluents, so unless we do something to prevent it, it can generate plans where $at(o, a) \wedge at(o, b)$ is true in some state.

We prune out these undesired states by adding suitable static laws. We do this while encoding the planning domain with \mathcal{B} . The language is not expressive enough for us to deduce these axioms automatically, and it is the responsibility of the programmer to find all relevant laws.

The problem of having one object at two different places at the same time can be resolved with static laws of the form:

$$\perp \text{ if } at(o, x) \wedge at(o, y) \wedge x \neq y .$$

10.5.3 Invalidating Preconditions

Suppose that we have a parallel action $\mathbf{A} = \{A_1, A_2\}$ where A_1 removes a precondition of A_2 . Then, \mathbf{A} is not a universal step since A_2 is not executable after we take A_1 . However, it is an existential step since we can take them in order $\langle A_2, A_1 \rangle$. If A_2 also removes a precondition of A_1 , then we do not have any linearization at all.

Example 10.5.2 *The VACUUM WORLD⁶ contains rooms and vacuum cleaners, and two of its actions are $clean(v, l)$ where the cleaner v cleans the room r and $move(v, x, y)$ where v moves from x to y .*

⁶See Section 10.9.1 for the complete description of VACUUM WORLD.

The preconditions of cleaning a room are that the cleaner is in the room and the room is dirty⁷ and the effect is that the room changes to not dirty.

Consider the parallel action:

$$\mathbf{A} = \{ \text{clean}(v, \text{kitchen}), \text{move}(v, \text{kitchen}, \text{hall}) \} .$$

Suppose that we execute it in a state:

$$S_1 = \{ \text{at}(v, \text{kitchen}), \neg \text{at}(v, \text{hall}), \text{dirty}(\text{kitchen}), \text{dirty}(\text{hall}) \}$$

executing \mathbf{A} leads us to the state:

$$S_2 = \{ \neg \text{at}(v, \text{hall}), \text{at}(v, \text{hall}), \neg \text{dirty}(\text{kitchen}), \text{dirty}(\text{hall}) \} .$$

There are two linearizations for \mathbf{A} :

$$\begin{aligned} \mathbf{A}_1 &= \langle \text{clean}(v, \text{kitchen}), \text{move}(v, \text{kitchen}, \text{hall}) \rangle \\ \mathbf{A}_2 &= \langle \text{move}(v, \text{kitchen}, \text{hall}), \text{clean}(v, \text{kitchen}) \rangle . \end{aligned}$$

Only \mathbf{A}_1 is valid since in \mathbf{A}_2 we try to clean a room where v is not located. Thus, \mathbf{A} is an existential step plan for cleaning the two rooms but it is not universal step.

In general, if we want that our plan is a universal step plan, we have to forbid parallel actions where this happens. In many planning domains we can do it automatically: if an action does not explicitly remove its precondition, we demand that it has to hold at the next state. This is essentially the same condition that Graphplan [9] uses for deciding whether two actions can be taken at the same time.

We can often do this enforcing by adding the precondition to the effects of the action. Thus, in VACUUM WORLD we demand that a cleaner stays in the room that it cleans and in BLOCKS WORLD we demand that $\neg \text{blocked}(a)$ holds after we take $\text{move}(a, b)$. In the next section we examine what to do when this approach does not work.

10.5.4 On Plans that cannot be Linearized

Just checking whether an action invalidates the precondition of another is enough for Graphplan since it has no static laws. However, with action languages it is possible that static laws rule out linearizations of some parallel actions.

Example 10.5.3 Consider the action description:

$$\begin{aligned} A_1 &\text{ causes } F_1 \text{ if } \neg F_1 \\ A_2 &\text{ causes } F_2 \text{ if } \neg F_2 \\ &\perp \text{ if } F_1 \wedge \neg F_2 \\ &\perp \text{ if } \neg F_1 \wedge F_2 . \end{aligned}$$

⁷In the complete description the cleaner also has to be plugged into a wall socket.

Both A_1 and A_2 are executable in $S = \{\neg F_1, \neg F_2\}$ and $E(\{A_1, A_2\}, S) = \{F_1, F_2\}$ that is a consistent set. However, the linearization $\langle A_1, A_2 \rangle$ fails because the first static law forbids states where F_1 is true and F_2 is not and the second static law rules out the linearization $\langle A_2, A_1 \rangle$ in the same way.

This behavior, in which we need to take a set of actions at the same time can be used to model synchronization in inherently parallel planning domains. However, it is undesirable if we want to find plans that can be linearized.

The problems arise when we have an action $\mathbf{A} = \{A_1, \dots, A_n\}$ such that $\langle S, \mathbf{A}, S' \rangle \in \mathbf{R}$ for some states S and S' but there is some subset $\mathbf{A}' \subset \mathbf{A}$ such that S has no successor with \mathbf{A}' .

We next describe three possible approaches that we can take:

1. we can work with a syntactically restricted subset of \mathcal{B} where every plan is guaranteed to be linearizable;
2. we add new fluents that force that problematic actions are taken sequentially; or
3. we can use a two-program construction where we first find a candidate parallel plan and then check if we can linearize its steps.

We will now examine these three conditions.

Syntactic Restrictions

The problem in Example 10.5.3 arose from the non-monotonicity of the static laws:

$$\begin{aligned} \perp & \text{ if } F_1 \wedge \neg F_2 \\ \perp & \text{ if } \neg F_1 \wedge F_2 . \end{aligned}$$

These laws are satisfied by⁸ $S_0 = \emptyset$ and $S_1 = \{F_1, F_2\}$ but not by any state S' , $S_0 \subset S' \subset S_1$.

This kind of a situation cannot happen if the static laws contain only positive fluents. For those action descriptions it holds that $Cn(E(\mathbf{A}, S)) \subseteq Cn(E(\mathbf{A}', S))$ whenever $\mathbf{A} \subseteq \mathbf{A}'$. So, if $Cn(E(\mathbf{A}, S))$ is inconsistent, then $Cn(E(\mathbf{A}', S))$ is also. Thus, we cannot have a situation where some subset of a consistent action is inconsistent.

Restricting the language to contain only positive fluents in static laws is a sufficient but not necessary condition to guarantee linearization. We will see this in the next section where we examine forced linearizations.

Forcing Linearization

Another approach is that we can identify which parallel actions $\mathbf{A} = \{A_1, \dots, A_n\}$ cause the linearization problem and then change the action description so that \mathbf{A} is no longer an executable action. This is a computationally heavy operation since in the worst case we have to examine every subset of \mathcal{A} to see whether it can cause the problem.

⁸Here we write only the positive literals of the states for clarity.

When we have identified such an action $\mathbf{A} = \{A_1, \dots, A_n\}$, we define a new fluent F_{ij} for each pair i and j where $1 \leq i < j \leq n$, and we add new dynamic laws:

$$\begin{aligned} A_i &\text{ causes } F_{ij} \\ A_j &\text{ causes } \neg F_{ij} \end{aligned}$$

for each fluent F_{ij} . Since now the set $E(\{A_i, A_j\}, S)$ is not consistent, we cannot take A_i and A_j at the same time.

Example 10.5.4 *Consider the action description:*

$$\begin{aligned} A_1 &\text{ causes } F_1 \text{ if } \neg F_1 \\ A_2 &\text{ causes } F_2 \text{ if } \neg F_2 \\ A_3 &\text{ causes } F_3 \text{ if } \neg F_3 \\ \perp &\text{ if } F_1 \wedge \neg F_2 \\ \perp &\text{ if } \neg F_1 \wedge F_2 . \end{aligned}$$

With three elementary actions we have seven possible non-empty parallel actions. Of those, the ones that contain both A_1 and A_2 are problematic but A_3 can be taken together with either of them. Thus, we modify the action description by adding the two new dynamic laws:

$$\begin{aligned} A_1 &\text{ causes } F_{12} \\ A_2 &\text{ causes } \neg F_{12} . \end{aligned}$$

Now, $\{A_1, A_2\}$ and $\{A_1, A_2, A_3\}$ are no longer executable actions since their effects are inconsistent..

This change is overly cautious because it always forces a complete linearization of the problematic elementary actions. For example, sometimes the problem manifests only if we take three or more actions at the same time but this construct forbids also taking two actions.

Note that we could do this same thing easier in the logic program transformation by adding the rules:

$$\leftarrow 2 \{ \text{tl}(A_1, i), \dots, \text{tl}(A_n, i) \}$$

for each problematic action $\{A_1, \dots, A_n\}$.

Checking for Linearization

The third option that we have is to use a two-program oracle construction in the same way that [23] uses it for conformant planning. Note that with this approach it is simpler to use existential step semantics instead of universal step.

We first use an **NP**-oracle to find an n -step parallel plan $\langle \mathbf{A}_1, \dots, \mathbf{A}_n \rangle$ that goes through states $\langle S_I, \dots, S_{n+1} \rangle$. Then, we use n calls to the oracle where we ask it to find a sequential plan from S_i to S_{i+1} using only the elementary actions from \mathbf{A}_i . Thus, we can find a sequential plan with $n + 1$ **NP**-queries [166].

Here we are transforming one large **NP**-query into a number of smaller **NP**-queries and we hope that solving the $n + 1$ small queries is easier than solving the large one.

If we want to use universal step semantics, we have to formulate the second question as: "Is it possible that we do not reach S_{i+1} with some ordering of the actions?" This translates the additional oracle calls into **co-NP**-queries.

10.6 INTRODUCING VARIABLES

In the language \mathcal{B} our fluents and actions are propositional atoms even though we used the predicate syntax in writing them. There is no reason why we should not make predicates an intrinsic feature of the language. We will now define a predicate version \mathcal{B}_P of \mathcal{B} . Our approach is similar to the one in [56].

Predicate Signatures

We will use typed predicates. We divide the elements of the universe into types and then assign a type to each argument position of a predicate symbol and say that only those elements that belong to that particular type can occur in that place. We will allow an element to belong to many types at the same time because that will make problem encoding easier.

Definition 10.6.1 *A predicate signature is a tuple $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$ where \mathcal{T} is a set of type identifiers, \mathcal{A} and \mathcal{F} are sets of predicate symbols,⁹ and f is a function that assigns a tuple $\langle t_1, \dots, t_n \rangle$ of type identifiers for every n -ary predicate symbol in $\mathcal{A} \cup \mathcal{F}$.*

A \mathcal{B}_P atom is of the form $p(V_1, \dots, V_n)$ where $p \in \mathcal{A} \cup \mathcal{F}$. An atom is an action atom if $p \in \mathcal{A}$ and a fluent atom if $p \in \mathcal{F}$.

Example 10.6.1 *The predicate signature for BLOCKS WORLD is*

$$\begin{aligned} \mathcal{S} &= \langle \mathcal{T}, \mathcal{A}, \mathcal{F} \rangle \\ \mathcal{T} &= \{object\} \\ \mathcal{A} &= \{move/2\} \\ \mathcal{F} &= \{on/2, blocked/1, is-block/1, equal/2\} \\ f &= \{move \mapsto \langle object, object \rangle, \\ &\quad on \mapsto \langle object, object \rangle, \\ &\quad blocked \mapsto \langle object \rangle \\ &\quad is-block \mapsto \langle object \rangle \\ &\quad equal \mapsto \langle object, object \rangle\} . \end{aligned}$$

The set of fluents is otherwise the same as before except that this time we have to add an explicit equality predicate. The simplest way to do so is to add it as a new fluent.

⁹We assume that the sets of action and fluent predicate symbols are disjoint.

Example 10.6.2 In the VACUUM WORLD we have to clean a multi-room apartment with a vacuum cleaner. Here we describe only the types and the actions of the domain. For the complete description of the domain see Section 10.9.1 on page 211. In the vacuum world we have three kinds of objects: cleaners, rooms, and power sockets and we define a type for all of them. The actions that are available to us are moving a cleaner to a different room, cleaning the room that the cleaner is in, and either plugging or unplugging a cleaner from a socket.

$$\begin{aligned}\mathcal{T} &= \{\text{cleaner}, \text{socket}, \text{room}\} \\ \mathcal{A} &= \{\text{move}/3, \text{clean}/2, \text{plug}/2, \text{unplug}/2\} \\ f &= \{\text{move} \mapsto \langle \text{cleaner}, \text{room}, \text{room} \rangle, \\ &\quad \text{clean} \mapsto \langle \text{cleaner}, \text{room} \rangle \\ &\quad \text{plug} \mapsto \langle \text{cleaner}, \text{socket} \rangle \\ &\quad \text{unplug} \mapsto \langle \text{cleaner}, \text{socket} \rangle\} .\end{aligned}$$

Action Descriptions

The \mathcal{B}_P action descriptions are defined in the same way as in simple \mathcal{B} .

Definition 10.6.2 (\mathcal{B}_P) A \mathcal{B}_P action description $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ is a triple consisting of a predicate signature $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$, a set \mathcal{L}_s of static laws, and a set \mathcal{L}_d of dynamic laws, where a static law is of the form

$$L \text{ if } F$$

where L is a fluent literal on \mathcal{S} and F is a conjunction of fluent literals, and a dynamic law is of the form

$$A \text{ causes } L \text{ if } F$$

where A is an action atom, L is a fluent atom and F is a conjunction of fluent literals.

An expression is either an atom (action or fluent), a literal, a conjunction of fluent literals, or a static or a dynamic law.

We do not have any constants in \mathcal{B}_P action descriptions and all terms are variables. The reason for this is that it makes definition of the semantics simpler as we do not have to worry about constant interpretation. However, we can simulate constants by using a suitable fluent that is true for only one element.

Example 10.6.3 When creating a \mathcal{B}_P encoding for the BLOCKS WORLD we can reuse the coding from Example 10.4.1 and replace the arguments of the predicates with variables. In addition, we have to add $\text{is-block}(X)$ as an additional precondition for $\text{move}(X, Y)$ so that we do not try to move the table. We also add a static law that prevents us from placing a block on top of itself. The complete encoding is shown in Figure 10.5.

$move(X, Y)$ **causes** $on(X, Y)$
 if $\neg blocked(X) \wedge \neg blocked(Y) \wedge is-block(X)$
 $move(X, Y)$ **causes** $\neg on(X, Z)$
 if $on(X, Z) \wedge \neg blocked(X) \wedge \neg blocked(Y) \wedge is-block(X)$
 $move(X, Y)$ **causes** $\neg blocked(X)$
 if $\neg blocked(X) \wedge \neg blocked(Y) \wedge is-block(X)$
 $\neg blocked(X)$ **if** $\neg is-block(X)$
 $blocked(X)$ **if** $on(Y, X) \wedge is-block(X)$
 \perp **if** $on(X, Y) \wedge on(Z, Y) \wedge \neg equal(X, Z) \wedge is-block(Y)$
 \perp **if** $on(X, X)$
 \perp **if** $on(X, Y) \wedge on(X, Z) \wedge \neg equal(Y, Z)$
 $equal(X, X)$ **if** \top
 $is-block(X)$ **if** $\neg equal(X, table)$

Figure 10.5: The action description for BLOCKS WORLD

Structures

An action description in \mathcal{B}_P acts as a uniform encoding for the particular planning domain. The individual planning instances are then defined in terms of structures where we define what objects belong to the types.

Definition 10.6.3 A \mathcal{B}_P -structure $\mathcal{I} = \langle \mathcal{S}, U, f_T \rangle$ consists of a predicate signature $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$, a set of elements U , and a function $f_T : \mathcal{T} \rightarrow 2^U$ that assigns a subset of U for every type identifier in \mathcal{S} . An \mathcal{I} -atom has the form $p(e_1, \dots, e_n)$ where $p \in \mathcal{A} \cup \mathcal{F}$, $f(p) = \langle T_1, \dots, T_n \rangle$, and for all $i \in [1, n]$ it holds that $e_i \in F_T(T_i)$.

Definition 10.6.4 A \mathcal{B}_P planning instance $\langle \mathcal{D}, \mathcal{I}, S_I, S_G \rangle$ consists of a \mathcal{B}_P action description, a initial state S_I and a goal condition S_G where S_I and S_G are sets of \mathcal{I} -fluent literals.

Example 10.6.4 Suppose that we have a BLOCKS WORLD instance with three blocks. Then, the corresponding structure is:

$$\begin{aligned}
 \mathcal{I} &= \langle \mathcal{S}, U, f_T \rangle \\
 U &= \{1, 2, 3, table\} \\
 f_T &= \{object \mapsto \{1, 2, 3, table\}\}
 \end{aligned}$$

where \mathcal{S} is as defined in Example 10.6.1.

Variable Types

Variables in actions and fluents are typed according to the argument positions where they occur. We will use a simple definition where each variable has a unique type in every expression that it appears in. We use this convention for simplicity even though having multiple types for a variable would allow us to eliminate fluents like $is-block/1$ in BLOCKS WORLD that act as type predicates.

Definition 10.6.5 Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$ be a \mathcal{B}_P action signature.

Then, the variable type assignment $T(V, E)$ of a variable V in an expression E is defined as follows:

1. If E is an atom $A = p(V_1, \dots, V_n)$ ($p \in \mathcal{A} \cup \mathcal{F}$), $T(V, A)$ is the set:

$$T(V, E) = \{T_i \mid f(p) = \langle T_1, \dots, T_n \rangle \text{ and } V = V_i \text{ for some } i \in [1, n]\}.$$

2. If E is a literal $\neg A$, then $T(V, E) = T(V, A)$.

3. If E is a conjunction of fluent literals $F_\wedge = F_1 \wedge \dots \wedge F_n$, then

$$T(V, E) = \bigcup_{i \in [1, n]} T(V, F_i) .$$

4. If E is a static law L **if** F , then

$$T(V, E) = T(V, L) \cup T(V, F) .$$

5. If E is a dynamic law A **causes** L **if** F_\wedge , then

$$T(V, E) = T(V, A) \cup T(V, L) \cup T(V, F_\wedge) .$$

An expression E is correctly typed if for every variable $V \in \text{Var}(E)$ it holds that $|T(V, E)| = 1$.

When E is correctly typed, $T(V, E)$ is a singleton set and we can treat it as a function that assigns the type to the variable.

Instantiation

A \mathcal{B}_P action description combined with a structure induces a planning domain. We define the domain via instantiation in the same way as we defined it for CCPs in Section 3.7.

Definition 10.6.6 Given a set of variables V and a function $f : V \rightarrow 2^U$ that associates a set of elements to each variable V , a \mathcal{B}_P -substitution σ assigns an element from $f(v)$ for every variable $v \in V$. The set of all possible substitutions with V and f is denoted by $\text{subs}(V, f)$.

A substitution σ applied to an atom $p(v_1, \dots, v_n)$ (where $\{v_1, \dots, v_n\} \subseteq V$) is the atom $p(\sigma(v_1), \dots, \sigma(v_n))$.

Definition 10.6.7 Let $\mathcal{I} = \langle \mathcal{S}, U, f_T \rangle$ be a \mathcal{B}_P structure and E be a correctly typed \mathcal{B}_P expression. Then, the \mathcal{I} -instantiation $\text{inst}_{\mathcal{I}}(F)$ is the set:

$$\{E\sigma \mid \sigma \in \text{subs}(\text{Var}(E), r_{\text{Var}(E)})\}$$

where $r_{\text{Var}(E)} : \text{Var}(E) \rightarrow 2^U$ is the function

$$r_{\text{Var}(E)}(V) = f_T(T(V, E)) .$$

The \mathcal{I} -instantiation of a \mathcal{B}_P action description $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ is the \mathcal{B} action description:

$$\text{inst}_{\mathcal{I}}(\mathcal{D}) = \langle \mathcal{S}', \mathcal{L}'_s, \mathcal{L}'_d \rangle$$

$$\mathcal{S}' = \{\{\text{inst}_{\mathcal{I}}(a(V_1, \dots, V_n)) \mid a \in \mathcal{A}\}, \{\text{inst}_{\mathcal{I}}(f(V_1, \dots, V_n)) \mid f \in \mathcal{F}\}\}$$

$$\mathcal{L}'_s = \{\text{inst}_{\mathcal{I}}(L) \mid L \in \mathcal{L}_s\}$$

$$\mathcal{L}'_d = \{\text{inst}_{\mathcal{I}}(L) \mid L \in \mathcal{L}_d\} .$$

Definition 10.6.8 A \mathcal{B}_P action description \mathcal{D} and a structure \mathcal{I} induce the LTS that corresponds to the instantiation $\text{inst}_{\mathcal{I}}(\mathcal{D})$.

10.6.1 Translating \mathcal{B}_P to ASP

We define the translation from \mathcal{B}_P to CCPs so that we can translate action descriptions and structures separately. The translation of an action description gives us then a uniform encoding for a planning domain and the structures are then defined with facts.

We can use essentially the same translation to CCPs as we did before, but we have to include the domain literals in the bodies of the rules. We will have two kinds of domain predicates, those representing the time and those representing types.

In these translations we again use the notation from Section 7 where an n -ary atom $p(V_1, \dots, V_n)$ is denoted by $p(\bar{V})$.

Representing Time

We will define a new domain predicate symbol $time/1$ that gives us all possible time steps. In all cases we define it by the rule:

$$time(1..t) \leftarrow$$

where t is an interpreted constant that evaluates to an integer.

There is one additional complication that we have to handle. If we can take t consequent actions, then we reach our final state at the time step $t + 1$ and we have to define our encoding so that the world state is consistent also then. Thus, we define also another time predicate $c-time/1$ that tells when the constraints have to hold:

$$c-time(1..t+1) \leftarrow .$$

We could do this also by imposing a restriction that all actions have to take place at a time step $i < t$. The two encodings are essentially equivalent and choosing between them is purely a matter of preference.

Definition 10.6.9 The program $time$ is a set of two facts:

$$time = \{ \langle time(1..t), \emptyset \rangle, \langle c-time(1..t+1), \emptyset \rangle \}$$

We will use the variable I exclusively for holding the time information in the translation.

Representing Type Information

We define a new unary domain predicate for every type that occurs in a predicate signature. Then, whenever a variable has that type in a law, we will add the correct domain literal for it.

Definition 10.6.10 Let \mathcal{T} be a set of type identifiers. Then, p_T denotes a unary type predicate symbol for each $T \in \mathcal{T}$.

Definition 10.6.11 Let E be a \mathcal{B}_P expression on a predicate signature $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$. Then, the set $\text{type}(E)$ of type literals is defined as follows:

$$\text{type}(E) = \{p_T(V) \mid V \in \text{Var}(E) \text{ and } T(V, E) = T\} .$$

The set $\text{type}(E)$ contains a domain literal for every variable V that occurs in E . The extensions of the type predicates will come from the \mathcal{B}_P structure that defines the particular planning instance.

Translating Atoms

We change an n -ary \mathcal{B}_P atom into a $n + 1$ -ary CCP atom by adding a new argument to hold the time information.

Definition 10.6.12 If $p(V_1, \dots, V_n)$ is an action atom, then the planning translation $\text{tl}(p(V_1, \dots, V_n)) = \text{action}_p(V_1, \dots, V_n, I)$. If $p(V_1, \dots, V_n)$ is a fluent atom, then the translations tl and tl_+ are defined as follows:

$$\begin{aligned} \text{tl}(p(V_1, \dots, V_n)) &= \text{fluent}_p(V_1, \dots, V_n, I) \\ \text{tl}_+(p(V_1, \dots, V_n)) &= \text{fluent}_p(V_1, \dots, V_n, I + 1) . \end{aligned}$$

For a negative fluent literal $\neg F$, $\text{tl}(\neg F) = \neg \text{tl}(F)$ and for a conjunction $F = F_1 \wedge \dots \wedge F_n$, $\text{tl}(F) = \bigcup_{i \in [1, n]} \text{tl}(F_i)$.

Again, we need rules to derive the strong negation of a fluent literal that is not true.

Definition 10.6.13 Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$ be a predicate signature. Then, the totalizing program $\text{total}(\mathcal{S})$ is the set of rules:

$$\begin{aligned} \text{total}(\mathcal{S}) = \{ \langle \neg \text{tl}(F), \{ \text{not } \text{tl}(F), c\text{-time}(I) \} \cup \text{type}(F) \rangle \mid \\ F \text{ is a fluent atom on } \mathcal{S} \} . \end{aligned}$$

Translating Static Laws

The static and dynamic laws are translated essentially the same way as we did earlier in the propositional case and we only need to add the domain literals to the rule bodies.

Definition 10.6.14 Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ be a \mathcal{B}_P action description. Then, the planning translation of a static law $S = L \text{ if } F \in \mathcal{L}_s$ is the rule:

$$\text{law}(S) = \langle \text{tl}(L), \text{tl}(F) \cup \text{type}(S) \cup \{c\text{-time}(I)\} \rangle .$$

The planning translation $\text{laws}(\mathcal{L}_s)$ is the set of rules:

$$\text{laws}(\mathcal{L}_s) = \bigcup_{S \in \mathcal{L}_s} \text{law}(S) .$$

Translating Dynamic Laws

Previously we defined the translations of preconditions and effects of an action in terms of the planning domain induced by it. Now that we want to have a uniform encoding we need to keep the variables in the atoms.

Definition 10.6.15 Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ be a \mathcal{B}_P action description. The precondition of an action $p(\bar{V})$ is the set:

$$\text{pre}(p(\bar{V})) = \bigcap_{p(\bar{V}) \text{ causes } L \text{ if } F \in \mathcal{L}_d} \{F_i \mid F = F_1 \wedge \dots \wedge F_n \text{ and } 1 \leq i \leq n\} .$$

The set of effects of an action is

$$\text{eff}(p(\bar{V})) = \{\langle L, F' \rangle \mid p(\bar{V}) \text{ causes } L \text{ if } F \in \mathcal{L}_d \text{ and } F' = F \setminus \text{pre}(A)\} .$$

Definition 10.6.16 Let $\mathcal{D} = \langle \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle, \mathcal{L}_s, \mathcal{L}_d \rangle$ be a \mathcal{B}_P action description. Then, the dynamic translation of an action $A = p(\bar{V}) \in \mathcal{A}$ is the set of rules:

$$\begin{aligned} \text{act}(A) = & \{\{\text{tl}(A)\}, \text{tl}(\text{pre}(A)) \cup dp(\{A\})\} \\ & \cup \{\text{tl}_+(L), \{\text{tl}(A)\} \cup \text{tl}(F) \cup dp(\{A, L, F\})\} \end{aligned}$$

where

$$dp(S) = \{\text{time}(I)\} \cup \bigcup_{L \in S} \text{type}(L) .$$

The translation $\text{acts}(\mathcal{L}_d)$ is the set of rules:

$$\text{acts}(\mathcal{L}_d) = \bigcup_{A \in \mathcal{A}} \text{act}(A) .$$

Inertia

The fluent inertia is defined in a similar way to Definition 10.4.2 except that we have to add the domain literals to the rule bodies.

Definition 10.6.17 Let $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$ be a predicate signature and $p \in \mathcal{F}$ be an n -ary predicate symbol. Then, the inertia translation $\text{inertia}(p)$ is the set of two rules:

$$\begin{aligned} \text{inertia}(p) = & \{\langle \text{tl}_+(p(\bar{V})), \{\text{tl}(p(\bar{V})), \text{not } \neg \text{tl}_+(p(\bar{V})), \text{time}(I)\} \\ & \cup \text{type}(p(\bar{V})) \rangle, \\ & \langle \neg \text{tl}(p(\bar{V})), \{\neg \text{tl}(p(\bar{V})), \text{not } \text{tl}_+(p(\bar{V})), \text{time}(I)\} \\ & \cup \text{type}(p(\bar{V})) \rangle\} \end{aligned}$$

The inertia translation $\text{inertia}(\mathcal{F})$ is the set

$$\text{inertia}(\mathcal{F}) = \bigcup_{p \in \mathcal{F}} \text{inertia}(p) .$$

Example 10.6.5 Suppose that we have an inertial fluent $at(X, Y)$ that is true when an object X is at the location Y . Then,

$$\begin{aligned} \text{tl}(at(X, Y)) &= \text{fluent}_{at}(X, Y, I) \\ \text{tl}_+(at(X, Y)) &= \text{fluent}_{at}(X, Y, I + 1) \\ \text{type}(at(X, Y)) &= \{\text{object}(X), \text{location}(Y)\} , \end{aligned}$$

and the rules of $\text{inertia}(at)$ are:

$$\begin{aligned} \text{fluent}_{at}(X, Y, I + 1) &\leftarrow \text{fluent}_{at}(X, Y, I), \text{not } \neg \text{fluent}_{at}(X, Y, I + 1), \\ &\quad \text{object}(X), \text{location}(Y), \text{time}(I) \\ \neg \text{fluent}_{at}(X, Y, I + 1) &\leftarrow \neg \text{fluent}_{at}(X, Y, I), \text{not } \text{fluent}_{at}(X, Y, I + 1), \\ &\quad \text{object}(X), \text{location}(Y), \text{time}(I) . \end{aligned}$$

Completing the Action Description Translation

The translation of a complete \mathcal{B}_P action description puts together all the component parts.

Definition 10.6.18 Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{L}_s, \mathcal{L}_d \rangle$ be a \mathcal{B}_P action description where $\mathcal{S} = \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle$. Then, the planning translation $\text{tl}(\mathcal{D})$ is the CCP:

$$\text{tl}(\mathcal{D}) = \text{time} \cup \text{laws}(\mathcal{L}_s) \cup \text{acts}(\mathcal{L}_d) \cup \text{inertia}(\mathcal{F}) \cup \text{total}(\mathcal{S}) .$$

Translating Structures and States

We get the extensions of the type predicates from a \mathcal{B}_P structure. We go through all types and add a fact for every element that occurs in a given type.

Definition 10.6.19 The domain translation $\text{domain}(\mathcal{I})$ of a \mathcal{B}_P structure $\mathcal{I} = \langle \langle \mathcal{T}, \mathcal{A}, \mathcal{F}, f \rangle, U, f_T \rangle$ is the ground CCP:

$$\text{domain}(\mathcal{I}) = \{ \langle p_T(e), \emptyset \rangle \mid t \in \mathcal{T} \text{ and } e \in f_T(t) \} .$$

When we have an initial state S_I , we set all literals that belong to it true at the first time step.

Definition 10.6.20 Let S_I be a set of literals. Then, the initial translation $\text{initial}(S_I)$ is the set of rules:

$$\begin{aligned} \text{initial}(S_I) &= \{ \langle \text{fluent}_p(e_1, \dots, e_n, 1), \emptyset \rangle \mid p(e_1, \dots, e_n) \in S_I \} \cup \\ &\quad \{ \langle \neg \text{fluent}_p(e_1, \dots, e_n, 1), \emptyset \rangle \mid \neg p(e_1, \dots, e_n) \in S_I \} . \end{aligned}$$

The literals that belong to the goal condition have to be true at the final time step.

Definition 10.6.21 Let S_G be a set of literals. Then, the goal translation is the set of rules:

$$\begin{aligned} \text{goal}(S_G) &= \{ \langle \perp, \{ \text{not } \text{fluent}_p(e_1, \dots, e_n, t + 1) \} \rangle \mid p(e_1, \dots, e_n) \in S_G \} \cup \\ &\quad \{ \langle \perp, \{ \text{not } \neg \text{fluent}_p(e_1, \dots, e_n, t + 1) \} \rangle \mid \neg p(e_1, \dots, e_n) \in S_G \} . \end{aligned}$$

Translating \mathcal{B}_P Instances

When we have a \mathcal{B}_P planning instance, we add the domain translation of the structure to the program.

Definition 10.6.22 *Let $\mathcal{I} = \langle \mathcal{D}, \mathcal{I}, S_I, S_G \rangle$ be a \mathcal{B}_P planning instance. Then, the translation $\text{tl}(\mathcal{D}, \mathcal{I}, S_I, S_G)$ is defined as:*

$$\text{tl}(\mathcal{I}) = \text{tl}(\mathcal{D}) \cup \text{domain}(\mathcal{I}) \cup \text{initial}(S_I) \cup \text{goal}(S_G) .$$

10.6.2 The Blocks World Example

Next we go through the complete BLOCKS WORLD encoding in \mathcal{B}_P and translate it to a CCP. We use the action description that we introduced previously in Figure 10.5.

Example 10.6.6 *The translation for the move action is:*

$$\begin{aligned} \{ \text{move}(X, Y, I) \} &\leftarrow \neg \text{blocked}(X, I), \neg \text{blocked}(Y, I), \\ &\quad \text{is-block}(X, I), \text{object}(X), \text{object}(Y), \text{time}(I) \\ \text{on}(X, Y, I + 1) &\leftarrow \text{move}(X, Y, I), \text{object}(X), \text{object}(Y), \text{time}(I) \\ \neg \text{on}(X, Z, I + 1) &\leftarrow \text{move}(X, Y, I), \text{on}(X, Z, I), \text{object}(X), \\ &\quad \text{object}(Y), \text{object}(Z), \text{time}(I) . \\ \neg \text{blocked}(X, I + 1) &\leftarrow \text{move}(X, Y, I), \text{object}(X), \\ &\quad \text{object}(Y), \text{time}(I) \end{aligned}$$

The translations for the static laws are:

$$\begin{aligned} \neg \text{blocked}(X, I) &\leftarrow \neg \text{is-block}(X, I), \text{object}(X), \text{c-time}(I) \\ \text{blocked}(X, I) &\leftarrow \text{on}(Y, X, I), \text{is-block}(X, I), \\ &\quad \text{object}(X), \text{object}(Y), \text{c-time}(I) \\ &\leftarrow \text{on}(X, Y, I), \text{on}(Z, Y, I), \neg \text{equal}(X, Z, I), \text{is-block}(Y, I), \\ &\quad \text{object}(X), \text{object}(Y), \text{c-time}(I) \\ &\leftarrow \text{on}(X, X, I), \text{object}(X), \text{c-time}(I) \\ &\leftarrow \text{on}(X, Y, I), \text{on}(X, Z, I), \neg \text{equal}(Y, Z, I), \\ &\quad \text{object}(X), \text{object}(Y), \text{object}(Z), \text{c-time}(I) \\ \text{equal}(X, X, I) &\leftarrow \text{object}(X), \text{c-time}(I) \end{aligned}$$

The inertia of fluents is translated to:

$$\begin{aligned}
on(X, Y, I + 1) &\leftarrow on(X, Y, I), \text{ not } \neg on(X, Y, I + 1), \\
&\quad object(X), object(Y), time(I) \\
\neg on(X, Y, I + 1) &\leftarrow \neg on(X, Y, I), \text{ not } on(X, Y, I + 1), \\
&\quad object(X), object(Y), time(I) \\
is-block(X, I + 1) &\leftarrow is-block(X, I), \text{ not } \neg is-block(X, I + 1), \\
&\quad object(X), time(I) \\
\neg is-block(X, I + 1) &\leftarrow \neg is-block(X, I), \text{ not } is-block(X, I + 1) \\
&\quad object(X), time(I) \\
blocked(X, I + 1) &\leftarrow blocked(X, I), \text{ not } \neg blocked(X, I + 1) \\
&\quad object(X), time(I) \\
\neg blocked(X, I + 1) &\leftarrow \neg blocked(X, I), \text{ not } blocked(X, I + 1) \\
&\quad object(X), time(I) \\
equal(X, X, I + 1) &\leftarrow equal(X, X, I), \text{ not } \neg equal(X, X, I + 1) \\
&\quad object(X), time(I) \\
\neg equal(X, X, I + 1) &\leftarrow \neg equal(X, X, I), \text{ not } equal(X, X, I + 1) \\
&\quad object(X), time(I) .
\end{aligned}$$

Finally, we add the time program and the totalizing program:

$$\begin{aligned}
time(1..t) &\leftarrow \\
c-time(1..t+1) &\leftarrow \\
\neg on(X, Y, I) &\leftarrow \text{ not } on(X, Y, I), object(X), object(Y), c-time(I) \\
\neg blocked(X, I) &\leftarrow \text{ not } blocked(X, I), object(X), c-time(I) \\
\neg is-block(X, I) &\leftarrow \text{ not } is-block(X, I), object(X), c-time(I) \\
\neg equal(X, I) &\leftarrow \text{ not } equal(X, I), object(X), c-time(I) .
\end{aligned}$$

When we look at this example, we see that it has many rules that contain redundant information. For example, we do not really need to add time information for the *is-block/1* fluent. If X is a block when we start the planning, it will always be a block. Even more redundant example is the fluent *equal/2* since we end up generating $4n \cdot (t + 1)$ rules where n is the number of objects and t the number of time steps when we could have added a simple test $X \neq Y$ to the rule bodies.

10.7 PLAN GENERATION

When we create a planning encoding using the previous schema, we end up with a program whose complexity is in **NP** because we use only a limited number of variables and only interpreted functions. The general PLANNING problem is **PSPACE**-complete [18] and we bridge this gap by using the oracle method.

An extremely naive algorithm for finding plans is shown in Figure 10.6. It starts with only one time steps and iteratively tests every possibility

```

function find-plan(Action Description  $A$ )
   $t := 1$ 
  while  $t \leq 2^{\|fluents\|}$  do
     $\langle R, M \rangle := \text{oracle}(\text{tl}(A, t))$ 
    if  $R = \text{sat}$  then
      return  $\text{actions}(M)$ 
    endif
     $t := t + 1$ 
  endwhile
  return  $\text{unsat}$ 
endfunction

```

Figure 10.6: A naive algorithm for plan generation

until it either finds a plan or it has determined that no plan exists. If there are n fluents in a planning instance, then there are at most 2^n different states. Since the longest possible plan visits every state once, we know that no plan may be longer than that. The function actions is an auxiliary that extracts the actions that we take from the answer set.

We can improve the exponential worst case behavior by finding the value of t with a binary search: first try find a plan with the length 2^{n-1} . If one exists, then try next with $t = 2^{n-2}$. If not, the next test is at $(2^{n-1} + 2^n)/2$. This approach drops the number of necessary queries to n .

If we are solving a real-world planning instance, we do not want to use the binary search even if it has a better worst-case behavior. A realistic instance can have hundreds or thousands of fluents and we cannot expect an ASP solver to handle a program with well over 2^{100} atoms in it.

A real-world heuristic is to start with an educated guess of a potential plan length, and then increase t by a fixed amount, usually between one and three, until either a plan is found or the instance sizes grow so large that they cannot be solved in a reasonable time.

10.8 MORE ON PLANNING VARIANTS

Next we examine some variants in both planning in general and action languages in particular. We keep our discussion mostly informal throughout this section.¹⁰

10.8.1 Fluent Variants

Many action languages allow us to define properties for fluents. We already saw how we can handle inertia that is the most important property. The properties that we consider are:

- *inertia*: an inertial fluent keeps its value if it is not explicitly changed by some action;

¹⁰We also leave domain predicates out of rule bodies.

- *default value*: a fluent can have a default value that it takes if it is not explicitly set to the other value;
- *freedom*: a free fluent can take any value during the planning if it is not explicitly set; and
- *fixedness*: a fixed fluent has the same value in every state of the domain.

Inertia

If a fluent is inertial, it keeps its value unless it is explicitly changed.

$$\begin{aligned} on(X, Y, I + 1) &\leftarrow on(X, Y, I), \text{ not } \neg on(X, Y, I + 1) \\ \neg on(X, Y, I + 1) &\leftarrow \neg on(X, Y, I), \text{ not } on(X, Y, I + 1) . \end{aligned}$$

Default Value

A fluent can have a default value associated for it. For example, a door or a gate can close automatically shortly after it is opened. We can model this behavior by adding a rule that allows us to conclude the default value unless its complement is explicitly set by some action:

$$closed(I) \leftarrow \text{ not } \neg closed(I) .$$

Free Fluents

A fluent is free if it may change its value freely during the planning. We can get this behavior by adding a choice rule for the fluent:

$$\{fluent(I)\} \leftarrow time(I) .$$

If we need also the strong negation of the free fluent, we can add a rule:

$$\neg true-fluent(I) \leftarrow \text{ not } true-fluent(I) .$$

Free fluents occur most often in dynamic planning domains where the state of the world may change independently of actions that are taken.

Fixed Fluents

Fixed fluents have the same truth value in all states of the planning domain. They are used to encode more fine-grained type information than what singly-typed variables allow. We will leave the time parameter out of them completely and we also define them so that they will be domain predicates. This allows us to cut down the size of the instantiation of the CCP.

For example, in BLOCKS WORLD we will define *is-block* directly in the instance description file with a set of facts of the form *is-block*(*x*) \leftarrow .

Default or Strong Negation

Using the strong negation in the encodings has the weakness that a pair of literals *F*, $\neg F$ is translated into two distinct atoms *F* and *F'* (Section 7.4.2) and a rule that forbids them to be true at the same time.

The strong negation is convenient for defining inertia but we can often do without when our planning formalism has non-inertial fluents.¹¹ For example, in BLOCKS WORLD we have the rule:

$$\{move(X, Y, I)\} \leftarrow \neg blocked(X, I), \neg blocked(Y, I) .$$

We derive a literal $\neg blocked(X, I)$ indirectly in terms of $blocked(X, I)$:

$$\begin{aligned} blocked(Y, I) &\leftarrow on(X, Y, I) \\ \neg blocked(Y, I) &\leftarrow \text{not } blocked(Y, I) . \end{aligned}$$

We can simplify the rules by cutting out $\neg blocked(X, I)$ out altogether:

$$\{move(X, Y, I)\} \leftarrow \text{not } blocked(X, I), \text{not } blocked(Y, I) .$$

From this on we use strong negation only with inertial fluents.

10.8.2 Action Variants

In this section we examine several variations for defining actions. We will keep our discussion on an informal level.

Complex Preconditions

Some action languages allow the user to write complex boolean expressions in preconditions or allow actions to have non-deterministic effects.

We can handle arbitrary preconditions for actions by using the same form as we used in our encoding for general Boolean expressions (Problem 7, page 161). For example, if the causation law is of the form:

$$a \text{ causes } e \text{ if } p_1 \vee p_2,$$

we can encode it with:

$$\begin{aligned} \{a\} &\leftarrow precondition_a \\ precondition_a &\leftarrow p_1 \\ precondition_a &\leftarrow p_2 . \end{aligned}$$

The remaining boolean connectives can be handled in same way. We can use conditional literals here to encode existential quantification. For example, we might want to say:

$$a(x) \text{ causes } e \text{ if } \exists y.p(x, y)$$

to denote that x may take the action a if there exists at least one y for which the precondition $p(x, y)$ holds. This can be encoded with:

$$\{a(X)\} \leftarrow 1 \{Y.p(X, Y) : domain(Y)\} .$$

¹¹With the exception of default values.

Executability Laws

Many action languages allow the user to write out the preconditions of an action explicitly with *executability laws* of the form:

$$\text{executable } A \text{ if } P .$$

The semantics is that A may be taken only if P is true. The simplest way how we can translate an executability law is to make P to be the precondition in the translation:

$$\{\text{tl}(A, I)\} \leftarrow \text{tl}(P, I) .$$

We can combine this translation with our original one to get a rule:

$$\{\text{tl}(A, I)\} \leftarrow \text{tl}(P, I), \text{tl}(P_A, I)$$

where P_A is the precondition that is computed from the dynamic laws of A in the manner of Definition 10.3.11.

A language can also have the complement construction **nonexecutable** that tells when an action can not be taken. A blocking rule can be implemented as a constraint. For example, the law:

$$\text{nonexecutable } a \text{ if } c$$

can be rendered with:

$$\leftarrow \text{tl}(a, i), \text{tl}(c, i) .$$

Temporal Considerations

The assumption that every action takes the same length makes planning much easier from the computational standpoint. Unfortunately, many real-world planning problems are inherently temporal and this simplification cannot be used with them.

As long as the time differences are not too long, we can say that the fastest action takes one time step to execute and the other actions take some fixed number of steps.

For example, if loading a package to a truck took three time steps, we could encode it as:

$$\begin{aligned} \{\text{load}(O, T, I)\} &\leftarrow \text{preconditions} \\ \text{in}(O, T, I + 3) &\leftarrow \text{load}(O, T, I) \\ \text{loading}(O, I_2) &\leftarrow \text{load}(O, T, I_1), I_1 \leq I_2 \leq I_1 + 3 . \end{aligned}$$

We define the predicate *loading/2* so that we can block conflicting actions from happening during loading.

This approach has the significant weakness that it makes the plans longer. The longer a plan is, the more difficult it is to compute.

10.9 TWO PLANNING EXAMPLES

Next we examine two additional planning domains. **VACUUM WORLD** is a simple example domain where we want to find an efficient way of

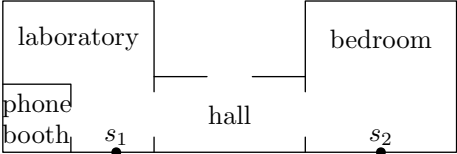
Instance	Plan
	$plug(c, s_1, 1) \quad move(c, lr, h, 7)$
	$clean(c, lr, 2) \quad plug(c, s_2, 8)$
	$move(c, lr, pb, 3) \quad clean(c, h, 9)$
	$clean(c, pb, 4) \quad move(c, h, br, 10)$
	$move(c, pb, lr, 5) \quad clean(c, br, 11)$
	$unplug(c, s_1, 6) \quad unplug(c, s_2, 12)$

Figure 10.7: A vacuum world example

cleaning an apartment, and we also encode the puzzle game Sokoban as a planning problem.

We give a \mathcal{B}_P encoding for VACUUM WORLD that uses some of the extensions from the previous section, and show how we can express it as rules. This domain is more complex than BLOCKS WORLD but it is still simple enough that we can give a straightforward description for it.

We can apply the same general principles to Sokoban but to get an encoding that is efficient enough to solve real game instances we have to apply domain-specific knowledge to the modeling. These optimizations are difficult to express in \mathcal{B}_P so we give only logic program rules for it.

10.9.1 Vacuum World

In the VACUUM WORLD [167] we have to clean a multi-room house with a vacuum cleaner. The otherwise simple domain is complicated by the fact that the cleaner has to be plugged in a power socket before it can be used and the power cord may not be long enough to reach every room from a single socket so we may have to change sockets during cleaning. Figure 10.7 shows a VACUUM WORLD instance and its solution and Table 10.1 contains a semi-formal description of the domain.

In this section we leave out the domain literals from the rule bodies. Instead, we use systematic naming for variables where each variable has always the same domain. For example, the variable C denotes always a vacuum cleaner so we add the domain literal $cleaner(C)$ to the bodies of all rules containing C .¹² The domain literals are shown in Table 10.1.

Fluents

The VACUUM WORLD has three inertial fluents that describe the basic facts of a state:

inertial $at(C, L)$
inertial $in\text{-}socket(C, S)$
inertial $dirty(L)$.

¹²This practice is supported by the *lpars* instantiator that uses `#domain` declarations to define implicit domain literals.

Fluents	
$at(c, l)$	The vacuum cleaner c is in room l . Inertial.
$dirty(l)$	The room l is dirty. Inertial.
$in-socket(c, s)$	The cleaner c is connected to the socket s . Inertial.
Auxiliaries	
$plugged(c)$	The cleaner c is plugged to some socket.
$reachable(l, s)$	Cleaner that is plugged in s can reach room l without unplugging.
$next(l_1, l_2)$	Rooms l_1 and l_2 are adjacent to each other
Actions	
$clean(c, l)$	Use cleaner c to clean the room l . Preconditions: $at(c, l)$, $dirty(l)$, and $plugged(c)$ Effects: $\neg dirty(l)$
$move(c, f, t)$	Move cleaner c from room f to t . Preconditions: $at(c, f)$, $next(c, f)$, either $\neg plugged(c)$ or $in-socket(c, s)$ and $reachable(t, s)$ Effects: $at(c, t)$, $\neg at(c, f)$
$plug(c, s)$	Plug the cleaner c into the socket s . Preconditions: $\neg plugged(c)$, $at(c, l)$ and $reachable(c, s)$ Effects: $in-socket(c, s)$.
$unplug(c, s)$	Unplug the cleaner c from the socket s . Preconditions: $in-socket(c, s)$ Effects: $\neg in-socket(c, s)$
Static Laws	
A cleaner may not be at two places at the same time.	
A cleaner may be plugged to only one socket at a time.	
A cleaner may take at most one action at each time step.	

Table 10.1: The VACUUM WORLD planning domain

Variable	Domain literal
C	$cleaner(C)$
L	$location(L)$
F	$location(F)$
T	$location(T)$
S	$socket(S)$
I	$time(I)$

Table 10.2: The domain literals of the VACUUM WORLD

Since we are not really interested in the negated versions of *at* and *in-socket*, we include only positive inertia rules for them:

$$\begin{aligned}
at(C, L, I + 1) &\leftarrow at(C, L, I), \text{ not } \neg at(C, L, I + 1) \\
in-socket(C, S, I + 1) &\leftarrow in-socket(C, S, I), \text{ not } \neg in-socket(C, S, I + 1) \\
dirty(L, I + 1) &\leftarrow dirty(L, I), \text{ not } \neg dirty(L, I + 1) \\
\neg dirty(L, I + 1) &\leftarrow \neg dirty(L, I) .
\end{aligned}$$

In the last rule we assume that there is not enough time for the rooms to become dirty again.

The layout of the apartment does not change during cleaning so we declare the fluents defining it fixed:

fixed *next*
fixed *reachable* .

In practice this means that we can use them as domain predicates if necessary.

We have also two auxiliary fluents, *plugged* and *may-move* that we describe later when we discuss the frame axioms.

Dynamic Laws

Clean We start with the action *clean*(C, L). Its laws are:

$$\begin{aligned}
clean(C, L) &\textbf{causes } \neg dirty(L) \textbf{ if } dirty(L) \wedge at(C, L) \wedge plugged(C) \\
clean(C, L) &\textbf{causes } at(C, L) \textbf{ if } dirty(L) \wedge at(C, L) \wedge plugged(C) \\
clean(C, L) &\textbf{causes } in-socket(C, S) \textbf{ if } \\
&dirty(L) \wedge at(C, L) \wedge plugged(C) \wedge in-socket(C, S) .
\end{aligned}$$

Even though $\neg dirty(L)$ is the only real effect of the action, we add *at*(C, L) and *in-socket*(C, S) to the effects to ensure that every cleaner does only one action in each time step; including *at*(C, L) prevents moving the cleaner from the room and *in-socket*(C, S) prevents unplugging it. The CCP rules for *clean* are:

$$\begin{aligned}
\{clean(C, L, I)\} &\leftarrow at(C, L, I), plugged(C, I), dirty(L, I) \\
\neg dirty(L, I + 1) &\leftarrow clean(C, L, I) \\
at(C, L, I + 1) &\leftarrow clean(C, L, I) \\
in-socket(C, S, I + 1) &\leftarrow clean(C, L, I), in-socket(C, S, I) .
\end{aligned}$$

$clean(C, L) \text{ causes } \neg dirty(L) \text{ if } dirty(L) \wedge at(C, L) \wedge plugged(C)$
 $clean(C, L) \text{ causes } at(C, L) \text{ if } dirty(L) \wedge at(C, L) \wedge plugged(C)$
 $clean(C, L) \text{ causes } in-socket(C, S) \text{ if } dirty(L) \wedge at(C, L)$
 $\quad \quad \quad \wedge plugged(C) \wedge in-socket(C, S)$
 $move(C, F, T) \text{ causes } at(C, T) \text{ if } at(C, F) \wedge may-move(C, F, T)$
 $move(C, F, T) \text{ causes } \neg at(C, F) \text{ if } at(C, F) \wedge may-move(C, F, T)$
 $move(C, F, T) \text{ causes } in-socket(C, S) \text{ if } in-socket(C, S)$
 $\quad \quad \quad \wedge may-move(C, F, T)$
 $plug(C, S) \text{ causes } in-socket(C, S) \text{ if } at(C, L) \wedge reachable(L, S)$
 $\quad \quad \quad \wedge \neg plugged(C)$
 $plug(C, S) \text{ causes } at(C, L) \text{ if } at(C, L) \wedge reachable(L, S)$
 $\quad \quad \quad \wedge \neg plugged(C)$
 $unplug(C, S) \text{ causes } \neg in-socket(C, S) \text{ if } in-socket(C, S)$
 $unplug(C, S) \text{ causes } \neg at(C, L) \text{ if } in-socket(C, S) \wedge at(C, L)$

Figure 10.8: The dynamic laws of VACUUM WORLD

Move There are two different ways to satisfy the precondition of the action *move*/3. We encode the preconditions using the boolean expression translation. However, we encode the disjunction with the help of static laws and an additional fluent *may-move* that is true when one of the preconditions is satisfied. This is clearer since otherwise the preconditions would become quite long

$move(C, F, T) \text{ causes } at(C, T) \text{ if } at(C, F) \wedge may-move(C, F, T)$
 $move(C, F, T) \text{ causes } \neg at(C, F) \text{ if } at(C, F) \wedge may-move(C, F, T)$
 $move(C, F, T) \text{ causes } in-socket(C, S) \text{ if } in-socket(C, S)$
 $\quad \quad \quad \wedge may-move(C, F, T) .$

The corresponding rules are:

$\{move(C, F, T, I)\} \leftarrow may-move(C, F, T, I)$
 $at(C, T, I + 1) \leftarrow move(C, F, T, I)$
 $\neg at(C, F, I + 1) \leftarrow move(C, F, T, I)$
 $in-socket(C, S, I + 1) \leftarrow move(C, F, T, I), in-socket(C, S, I)$

The disjunctive precondition is defined as:

$may-move(C, F, T) \text{ if } at(C, F) \wedge next(F, T) \wedge$
 $\quad \quad \quad (\neg plugged(C) \vee (in-socket(C, S) \wedge reachable(S, T)))$

When expressed as rules it becomes:

$may-move(C, F, T, I) \leftarrow at(C, F, I), next(F, T), \text{not } plugged(C, I)$
 $may-move(C, F, T, I) \leftarrow at(C, F, I), next(F, T), in-socket(C, S, I),$
 $\quad \quad \quad reachable(T, S) .$

$$\begin{aligned}
\text{may-move}(C, F, T) \text{ if } & \text{at}(C, F) \wedge \text{next}(F, T) \wedge \\
& (\neg \text{plugged}(C) \vee (\text{in-socket}(C, S) \wedge \text{reachable}(S, T))) \\
\text{plugged}(C) \text{ if } & \text{in-socket}(C, S) \\
\perp \text{ if } & \text{at}(C, L_1) \wedge \text{at}(C, L_2) \wedge L_1 \neq L_2 \\
\perp \text{ if } & \text{in-socket}(C, S_1) \wedge \text{in-socket}(C, S_2) \wedge S_1 \neq S_2
\end{aligned}$$

Figure 10.9: The static laws of VACUUM WORLD

Plug A vacuum cleaner may be plugged in if it is within reach of a socket:

$$\begin{aligned}
\text{plug}(C, S) \text{ causes } \text{in-socket}(C, S) \text{ if } & \text{at}(C, L) \wedge \text{reachable}(L, S) \\
& \wedge \neg \text{plugged}(C) \\
\text{plug}(C, S) \text{ causes } \text{at}(C, L) \text{ if } & \text{at}(C, L) \wedge \text{reachable}(L, S) \\
& \wedge \neg \text{plugged}(C) .
\end{aligned}$$

The rules for this action are:

$$\begin{aligned}
\{\text{plug}(C, S, I)\} & \leftarrow \text{at}(C, L, I), \text{reachable}(L, S), \text{not } \text{plugged}(L, I) \\
\text{in-socket}(C, S, I+1) & \leftarrow \text{plug}(C, S, I) \\
\text{at}(C, L, I+1) & \leftarrow \text{plug}(C, S, I), \text{at}(C, L, I) .
\end{aligned}$$

Unplug Unplugging a cleaner is straightforward:

$$\text{unplug}(C, S) \text{ causes } \neg \text{in-socket}(C, S) \text{ if } \text{in-socket}(C, S)$$

The rules for it are:

$$\begin{aligned}
\{\text{unplug}(C, S, I)\} & \leftarrow \text{in-socket}(C, S, I) \\
\neg \text{in-socket}(C, S, I+1) & \leftarrow \text{unplug}(C, S, I) .
\end{aligned}$$

Static Laws

The predicate $\text{plugged}/2$ is an auxiliary that is defined with the rule:

$$\text{plugged}(C, I) \leftarrow \text{in-socket}(C, S, I) .$$

Next, we need frame axioms to forbid the cleaner from being in two places at the same time and from being plugged to two sockets at one time.

$$\begin{aligned}
& \leftarrow 2 \{L.\text{at}(C, L, I) : \text{location}(L)\} \\
& \leftarrow 2 \{S.\text{in-socket}(C, S, I) : \text{socket}(S)\} .
\end{aligned}$$

Finally, we demand that at the end all rooms have to be clear and all vacuum cleaners unplugged:¹³

$$\begin{aligned}
& \leftarrow \text{dirty}(L, t+1), \text{location}(L) \\
& \leftarrow \text{plugged}(C, t+1), \text{cleaner}(C) .
\end{aligned}$$

¹³Here we remove the double negation from not $\neg L$ and use L , instead.

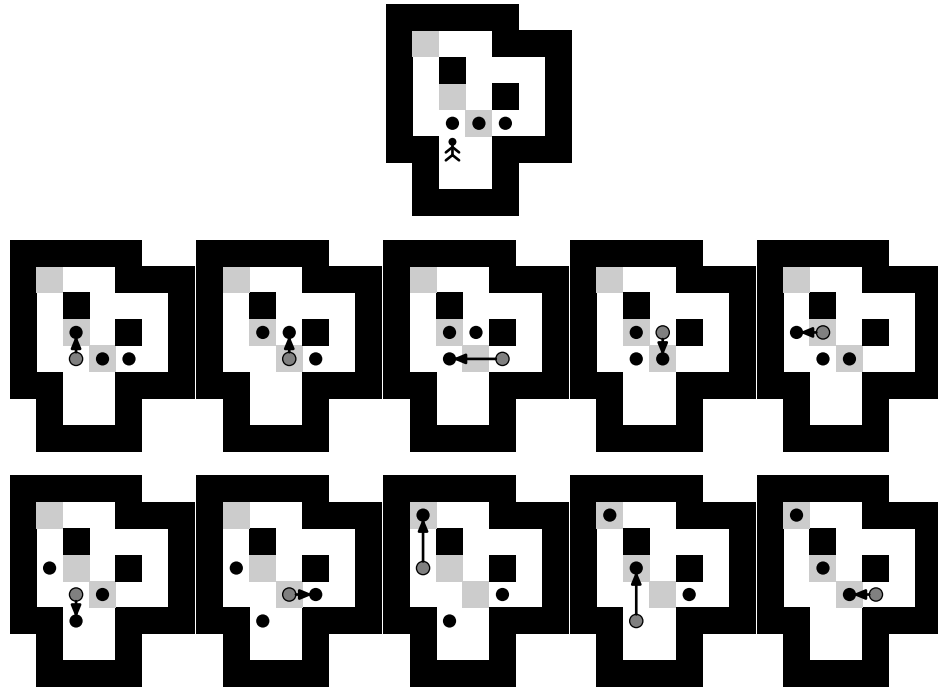


Figure 10.10: A Sokoban instance and an optimal solution

10.9.2 Sokoban

In the game of Sokoban,¹⁴ the player takes the role of a warehouse keeper (“Sokoban”) who has to rearrange a number of boxes. Sokoban can move boxes only by pushing them in one of the four main directions and he cannot climb over or squeeze between them. One example Sokoban puzzle¹⁵ and its solution is shown in Figure 10.10.

In this section we create a planning-based encoding for solving Sokoban puzzles. Since Sokoban games may be rather large, we have to pay close attention in making our solution as efficient as possible.

How to Model Sokoban Puzzles?

The first step in modeling Sokoban is to decide what a move will be. One possibility would be to use Sokoban’s movement directly, so that one time step of the plan corresponds to Sokoban moving one square in the puzzle. However, this approach has the problem that the plans quickly become dozens or hundreds of steps long, far too long to be solved in a reasonable amount of time. Instead, we take the approach that a move begins when Sokoban starts to push a box into a direction, and ends when the box finally comes to rest and Sokoban starts to move another box. Thus, if a box is pushed four squares in a row, it is taken to be a single move.

Sokoban *move* is the most complex action that we have seen up to date. With an abstracted action language like syntax it becomes:

¹⁴The game of Sokoban was originally created by Hiroyuki Imabayashi in 1980.

¹⁵This level is designed by Yoshio Murase and it is downloaded from <http://www.ne.jp/asahi/ai/yoshio/sokoban/main.htm>.

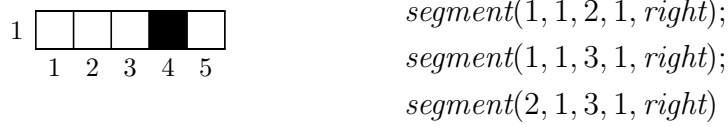


Figure 10.11: An example of Sokoban segments

$move(x_1, y_1, x_2, y_2)$ **causes** $has-box(x_2, y_2) \wedge at(x_3, y_3) \wedge \neg has-box(x_1, y_1)$
if $has-box(x_1, y_1);$
 (x_1, y_1) and (x_2, y_2) lie on the same segment;
the squares between (x_1, y_1) and (x_2, y_2) are empty;
Sokoban can reach the square behind (x_1, y_1) ; and
the last square Sokoban enters is (x_3, y_3) .

We can divide the preconditions into dynamic and static preconditions where a static property depends only on the description of the problem instance and a dynamic property depends on how conditions change during planning. Essentially, fluents that describe static preconditions have fixed values that cannot change during the planning.

Here the only static precondition is the $segment(x_1, y_1, x_2, y_2)$ that denotes that the squares (x_1, y_1) and (x_2, y_2) are in the same line segment and we can push a box from the former to the latter in one movement action. We can automatically prune out all model candidates that violate this property by making it a domain predicate and creating only those instances of $move$ action that satisfy it.

For example, in the grid example shown in Figure 10.11 we see that $(2, 1)$ is on the same segment as $(3, 1)$ but the wall in $(4, 1)$ blocks $(5, 1)$ from being on it.

One of the dynamic preconditions for $move$ is that Sokoban has to be able to reach the position that is immediately behind the box that is to be moved. This means that we have to compute the set of all squares that Sokoban can reach at each time step. This is a feature that makes Sokoban tricky to model using planning formalisms that are based on propositional logic [106] since expressing the transitive closure of a reachability relation is not straightforward.¹⁶ Also, note that the reachability condition is partly static. In Figure 10.11 we note that even though $(1, 1)$ and $(3, 1)$ lie on the same segment, it is actually not possible to push a box from one to the other since the walls block Sokoban from getting behind the boxes. We can again use domain predicates to remove these impossible moves from the encoding.

The final consideration about the $move$ action is that we have large numbers of actions that have similar preconditions and effects. The only thing that separates actions $move(1, 1, 1, 4)$ and $move(1, 1, 1, 5)$ is that the

¹⁶One way to do it with only a logarithmic increase on the number of atoms can be adapted from [101].

latter requires that $(1, 5)$ is also empty but otherwise their preconditions are exactly the same.

We can reduce the instantiation sizes of our problems by making use of this similarity by breaking the *move/4* action into two predicate symbols: *move-from* (X, Y, D) and *move-to* (X, Y) where the first one encodes the starting location and the direction of a move, and the second encodes the target location.

To see how much this helps consider the case where we have a segment that is 10 squares long. With a single *move* action and disregarding most of the preconditions and effects our rules have the form:

$$\begin{aligned} \{move(X_1, Y_1, X_2, Y_2, I)\} &\leftarrow segment(X_1, Y_1, X_2, Y_2) \\ has-box(X_2, Y_2, I + 1) &\leftarrow move(X_1, Y_1, X_2, Y_2, I) \\ \neg has-box(X_1, Y_1, I + 1) &\leftarrow move(X_1, Y_1, X_2, Y_2, I) \end{aligned}$$

With 10 squares in the segment, there are 90 possible move actions when we suppose that we can traverse the segment fully in both directions. This means that all three rules have 90 instances with a total of 270 ground rules.

On the other hand, if we split the rules we get:

$$\begin{aligned} \{move-from(X, Y, D, I)\} &\leftarrow possible-push(X, Y, D) \\ 1 \{ \{X_2, Y_2\}.move-to(X_2, Y_2, I) : segment(X_1, Y_1, X_2, Y_2, D) \} &1 \\ &\leftarrow move-from(X_1, Y_1, D) \\ \neg has-box(X, Y, I + 1) &\leftarrow move-from(X, Y, D, I) \\ has-box(X, Y, I + 1) &\leftarrow move-to(X, Y, I) . \end{aligned}$$

Splitting the *move* action made it necessary to add the directional parameter to *segment*. Now the first three rules have 18 ground instances where the second rule has on average five literals inside the choice. The final rule has 10 instances so the total for this encoding is 64 ground instances, a saving of about 75%.

Instance Description

The Sokoban domain differs from our previous planning domains in that we want to define a number of domain predicates that will help us to make the encoding more efficient in practice. Our data predicates will be:

- *initial-at* (x, y) : the initial location of Sokoban;
- *initial-box* (x, y) : the initial positions of boxes;
- *square* (x, y) : locations of open squares; and
- *target-square* (x, y) : the goal locations.

The new domain predicates are:

- *m-square* (x, y) : this predicate encodes all possible locations where a box may be during the planning;

- $segment(x_1, y_1, x_2, y_2, d)$: starting at location (x_1, y_1) there is an uninterrupted straight line to (x_2, y_2) along the direction d ;
- $between(x, y, x_1, y_1, x_2, y_2)$: the square (x, y) lies between squares (x_1, y_1) and (x_2, y_2) ; and
- $possible-push(x, y, d)$: it is possible that some time during the planning we may push a box from the location (x, y) to the direction d .

The idea behind $m-square/2$ is that we know from outset that there are some moves that we may not take during the planning and we want to rule them out immediately. For example, if we push a box into a corner, it is stuck in there. If the corner is not a goal square, we have made the puzzle unsolvable.

We define $m-square/2$ using an auxiliary predicate $has-route(x, y)$ that is true if it is possible to push a box from (x, y) to some goal location. It is defined with the rules:

$$\begin{aligned} has-route(X, Y) &\leftarrow target-square(X, Y) \\ has-route(X, Y) &\leftarrow has-route(X + 1, Y), \\ &\quad square(X - 1, Y), square(X, Y) . \end{aligned}$$

The second rule states that if we can push a box to right from (x, y) and reach a square that has a route to some target, then we have a route from (x, y) . In addition we need three corresponding rules for the other directions. After that we can define $m-square/2$:

$$m-square(X, Y) \leftarrow has-route(X, Y) .$$

The other three new domain predicates are defined in terms of $m-square$. The $segment$ is defined with rules of the form:

$$\begin{aligned} segment(X, Y, X + 1, Y, right) &\leftarrow m-square(X, Y), m-square(X + 1, Y) \\ segment(X_1, Y, X_2, Y, right) &\leftarrow segment(X_1, Y, X_2, Y, right), \\ &\quad m-square(X_2 + 1, Y) . \end{aligned}$$

The predicate $possible-push/3$ is defined with rules of the form:

$$\begin{aligned} possible-push(X, Y, right) &\leftarrow m-square(X, Y), m-square(X + 1, Y), \\ &\quad square(X - 1, Y) . \end{aligned}$$

Finally, $between/6$ is defined with four rules of the form:

$$\begin{aligned} between(X, Y, X_1, Y, X_2, Y) &\leftarrow segment(X_1, Y, X_2, Y, right), \\ &\quad square(X, Y), \\ &\quad X > X_1, X \leq X_2 . \end{aligned}$$

The reason why we use $X \leq X_2$ instead of $X < X_2$ is that we can then use one rule to check that both the target square and all squares leading to it are empty.

Fluents

The two most important fluents are:

- $has\text{-}box(x, y)$: there is a box at location (x, y) ; and
- $at(x, y)$: Sokoban is at location (x, y) .

The box location is inertial while at is not since Sokoban makes a move every time step.

We also define several auxiliary fluents whose values are computed from the previous fluents and instance description:

- $can\text{-}push(x, y, d)$: it is possible to push the box that is in location (x, y) to the direction d . This fluent is used to abstract the preconditions of the *move* action.
- $reachable(x, y)$: Sokoban can reach the location (x, y) .

We can make a push if there is a box in the location and the location behind it is reachable. We again use four rules, one for each direction:

$$\begin{aligned} can\text{-}push(X, Y, right, I) \leftarrow & possible\text{-}push(X, Y, D), \\ & has\text{-}box(X, Y, I), \\ & reachable(X - 1, Y, I) . \end{aligned}$$

The definition of $reachable/2$ computes a transitive closure of the adjacency relation from the current location through the empty squares:

$$\begin{aligned} reachable(X, Y, I) \leftarrow & at(X, Y, I) \\ reachable(X + 1, Y, I) \leftarrow & reachable(X, Y, I), \\ & square(X + 1, Y), square(X, Y), \\ & not\ has\text{-}box(X + 1, Y) . \end{aligned}$$

Since $has\text{-}box$ was inertial, we need the rule:

$$has\text{-}box(X, Y, I + 1) \leftarrow has\text{-}box(X, Y, I), not\ \neg has\text{-}box(X, Y, I + 1) .$$

The values for the fluents are initialized from the data predicates:

$$\begin{aligned} has\text{-}box(X, Y, 1) \leftarrow & initial\text{-}box(X, Y) \\ at(X, Y, 1) \leftarrow & initial\text{-}at(X, Y) . \end{aligned}$$

Actions

The *move/4* action is split into *move-from/3* and *move-to/2* actions:

$$\begin{aligned} \{move\text{-}from(X, Y, D, I)\} \leftarrow & can\text{-}push(X, Y, D, I), \\ & possible\text{-}push(X, Y, D) \\ 1\ \{\{X_2, Y_2\}.move\text{-}to(X_2, Y_2, I) : \\ segment(X_1, Y_1, X_2, Y_2, D)\} \ 1 \leftarrow & move\text{-}from(X_1, Y_1, D), \\ & possible\text{-}push(X_1, Y_1, D) . \end{aligned}$$

These rules do not take the requirement that the boxes have to pass through empty squares into account. We add it as a constraint:

$$\begin{aligned} \leftarrow & \text{has-box}(X, Y, I), \text{between}(X, Y, X_1, Y_1, X_2, Y_2), \\ & \text{move-from}(X, Y, D, I), \text{move-to}(X_2, Y_2, I), \\ & \text{segment}(X_1, Y_1, X_2, Y_2, D). \end{aligned}$$

The main effect is that the box moves to the new location:

$$\begin{aligned} \text{has-box}(X, Y, I + 1) & \leftarrow \text{move-to}(X, Y, I) \\ \neg \text{has-box}(X, Y, I + 1) & \leftarrow \text{move-from}(X, Y, I), \\ & \text{possible-push}(X, Y, D, I) . \end{aligned}$$

The other effect is that the location of Sokoban also changes. We can simplify the rules for *at* by noting that the location where we started pushing is necessarily reachable from Sokoban's new location. Since Sokoban can move freely on the empty squares, we can take this square as Sokoban's new location without causing an error:

$$\text{at}(X, Y, I + 1) \leftarrow \text{move-from}(X, Y, D, I), \text{possible-push}(X, Y, D) .$$

Consistency Constraints

Because in this domain the boxes are functionally equivalent to each other, we do not have to worry about them being in multiple locations at the same time. In fact, the only consistency constraint that we need is a rule that forbids us from taking two moves in one time step:

$$\leftarrow 2 \{ \{X, Y\}. \text{move-to}(X, Y, I) : m\text{-square}(X, Y) \} .$$

Optimizations

We can add a number of other optimizations to make the plan generation more efficient. We consider two basic kinds of optimizations:

1. optimizations that prune out incorrect branches of the search tree; and
2. optimizations that prune redundant moves from the plan.

For example, we know that Sokoban should never push two boxes together along a wall since they will be stuck there. The only exception is if both of the squares are goal locations. This constraint can be encoded with rules of the form:

$$\begin{aligned} \text{edge-pair}(X, Y, X + 1, Y) & \leftarrow m\text{-square}(X, Y), m\text{-square}(X + 1, Y), \\ & \text{not square}(X, Y + 1), \\ & \text{not square}(X + 1, Y + 1) \\ \text{allowed-pair}(X, Y, X + 1, Y) & \leftarrow \text{target-square}(X, Y), \\ & \text{target-square}(X + 1, Y) \\ & \leftarrow \text{edge-pair}(X_1, Y_1, X_2, Y_2), \\ & \text{not allowed-pair}(X_1, Y_1, X_2, Y_2), \\ & \text{has-box}(X_1, Y_1, I), \\ & \text{has-box}(X_2, Y_2, I) . \end{aligned}$$

We can use similar rules to remove plan candidates where three boxes are pushed to an *L*-shape around a corner or four boxes pushed to form a square.

An example of the second kind of optimization is that we can forbid Sokoban from making two moves where one would be sufficient. We make it an error if one box is pushed two times to the same direction:

$$\begin{aligned} &\leftarrow \text{push-dir}(D, I), \\ &\quad \text{move-to}(X, Y, I) \\ &\quad \text{move-from}(X, Y, D, I + 1) \\ \text{push-dir}(D, I) &\leftarrow \text{move-from}(X, Y, D, I) . \end{aligned}$$

Another similar optimization is that we might explicitly forbid Sokoban from immediately reversing a move. These conditions can be expressed as:

$$\begin{aligned} &\leftarrow \text{push-dir}(\text{right}, I), \\ &\quad \text{move-to}(X, Y, I) \\ &\quad \text{move-from}(X, Y, \text{left}, I + 1) \\ &\quad \text{reachable}(X + 1, Y, I) . \end{aligned}$$

We add $\text{reachable}(X + 1, Y, I)$ to the rule body because we might need to push a box back to the same direction if we were not able to go to the other side previously.

The final possible consideration is that we can further reduce the size of the ground instantiation with additional domain predicates. Since we know where all boxes are at the beginning, we need not generate rules for squares that we know are empty. To do this we can add a new domain literal $\text{possible-box}(X, Y, I)$ to every rule where a literal $\text{has-box}(X, Y, I)$ occurs somewhere. A reasonably simple definition for it could be:

$$\begin{aligned} \text{possible-box}(X, Y, 1) &\leftarrow \text{initial-box}(X, Y) \\ \text{possible-box}(X, Y, 2) &\leftarrow \text{initial-box}(X, Y) \\ \text{possible-box}(X_2, Y_2, 2) &\leftarrow \text{initial-box}(X_1, Y_1), \\ &\quad \text{segment}(X_1, Y_1, X_2, Y_2, D) \\ \text{possible-box}(X, Y, I) &\leftarrow \text{m-square}(X, Y), I > 2 . \end{aligned}$$

Here a box may be only in its initial location in the first time step and be somewhere along the same segment in the next time step. As keeping track of possible locations gets complicated quite soon, we stop it at the third time step and suppose that the boxes can be anywhere in the area.

11 CONCLUSIONS

The cardinality constraint programs are a powerful extension of normal logic programs. The cardinality and conditional literals, together with choice rules make it possible to have intuitive and concise uniform answer set programming encodings for many if not most **NP**-complete problems. With the oracle method we can use these **NP**-encodings to solve also many **PSPACE**-complete problems.

Cardinality literals allow us to express conditions that are based on sets of objects, and conditional literals make it possible to define those sets concisely. Choice rules help in creating uniform encodings since they facilitate the generate-and-test method by allowing a clear separation of the generator and the tester.

We define the language of cardinality constraint programs in two phases where the semantics is first defined for a basic language and the extended constructs of the full language are defined as translations to the basic language. This approach allows us to have both a simple semantics that is easy to understand and implement and an expressive language for writing problem encodings. The semantics itself is based on the stable model semantics of normal logic programs.

A major criterion on the design of the language is that it should be possible to do the translation from the full language to the basic language on the level of programs with variables. This makes it easier to incorporate modularity to programming. We want to use uniform encodings and usually we do not know what particular problem instances we want to solve. When the translation does not depend on the particular instance, we can do it immediately after we create the encoding and before we have the instances. Also, if we want to try to use some different encoding for a subproblem, we can do the change without having to worry about the facts of the instance description.

Another consideration was that the transformations should not add new atoms to the programs. This goal was not completely satisfied, but the new atoms that we generate have the property that it is possible to compute their extensions in linear time. This means that they do not significantly increase the computational effort for finding answer sets.

The published literature contains many other types of aggregate literals in addition to cardinality atoms. The reason why we selected only one aggregate into the language is simplicity: cardinality atoms are monotonic and adding them does not increase the computational complexity of the semantics. Using only monotonic aggregates has the advantage that the definition of the stable model semantics is simple and intuitive, and it can be done using a provability-operator analogously to the normal programs. The downside is that it is not possible to use nonmonotonic aggregates with this approach.

However, we have been able to create reasonably concise uniform encodings for most of the graph problems [198] in the classic list of **NP**-complete problems compiled by Garey and Johnson [78]. The most significant weakness is that even with weight constraints it is cumbersome to

perform arithmetic with numbers that occur as arguments in an arbitrary set of atoms.

We define the ω -restricted subclass of the language that has the property that it stays decidable even when we allow function symbols with the Herbrand interpretation. We create a hierarchy of predicate symbols where the first stratum has a finite extension and each new stratum adds a finite number of atoms into the answer set.

The computational complexity of for deciding whether a ω -restricted program has an answer set is in **2-NEXP** and thus computationally extremely intractable. Fortunately, most practical encodings do not need the full power of the language. The most significant raise in complexity comes from function symbols with the Herbrand interpretation. When we do not use such functions, the complexity of the language is the same as the stable model semantics for normal logic programs. In practice, most of our encodings will use only few variables and all functions are arithmetic operators so they belong to the subclass where deciding whether an answer set exists is **NP**-complete.

We also examined briefly the question on efficiency of encodings. It is possible that we have two programs that are strongly equivalent in the sense that they have the same answer sets no matter the context where we use them but where we need much more effort to solve one than the other. We gave a practical example where we optimized an encoding for solving Kakuro puzzles. We tested the encodings with an extremely large puzzle instance and the result was that two solvers, *smodels* and *clasp*, could find a solution and prove that it was unique without making any guesses at all when the optimizations were used, while with the straightforward encoding *smodels* needed almost 600 guesses to find a solution and almost 1000 guesses to prove that it was the only one and for *clasp* the figures were 7248 and 7256, respectively.

The *lparse* instantiator handles a large subset of ω -restricted CCPs under the standard interpretation. We described the general structure of the implementation and compared it with a compiler for an ordinary programming language. The front-end of *lparse* reads in a user program written in the full language and it translates it into an internal representation which is almost equivalent to the basic language. Then, the back-end does the actual instantiation. This basic architecture allows us to use the same framework to instantiate different logical languages by writing a new front-end and also to use different semantics for one program by changing the back-end to produce a different target language.

The actual instantiation works in two phases. First we compute the domain model and then use it to instantiate the non-domain program. The instantiator handles all rules in a uniform way, so when we compute the model of a stratum program we first instantiate it normally and then compute its least model.

As the final part of this work we examined how we can use ASP techniques in general and CCPs in particular to solve AI planning problems. Many real-world problems can be expressed as planning problems so it is an important research area. The advantage of ASP in planning is that we can have declarative programs that are simple to understand.

We present a systematic translation from an action language into CCPs and showed that it is faithful to the original semantics. We identify a set of possible sources of problems that we have to consider when making encodings that admit serializable parallel planning, since there is a risk that if we are not careful we might end up with plans that cannot be realized in practice.

We also showed how we can use domain-specific knowledge to obtain encodings that are more efficient in practice. Planning problems are computationally hard and if we want to be able to solve real-world problems, we have to expend significant effort to make the system fast enough.

In conclusion, this work examined both theoretical and practical aspects of answer set programming. We defined a language and analyzed its theoretical properties and then showed how we can solve **NP**- and **PSPACE**-complete problems with it.

Aggregates are a major focus on the current research on the theoretical aspects of ASP. A number of different approaches has been proposed for them and none of them has gained prominence over others. Finding a semantics that gives answer sets that seem intuitively correct and that does not increase the computational complexity is a difficult problem that can be hopefully solved in near future.

From the practical side one of the major challenges is to develop frameworks that allow simpler interfacing between ASP and traditional programs. An ASP solver might be able to find answers for all of our interesting instances of an **NP**-complete problem, but that does not help the user if there is no convenient way to create the instance facts or interpret the answers. ASP is still mostly a field for scientific research. To change that and bring ASP into the public consciousness we need a better integration between solvers and conventional programming.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [2] Christian Anger, Kathrin Konczak, and Thomas Linke. Nomore: A system for non-monotonic reasoning under answer set semantics. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 406–410, September 2001.
- [3] Krzysztof R. Apt and Roland N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19–20:9–71, 1994.
- [4] Krzysztof R. Apt and Maarten H van Emden. Contributions to the theory of logic programming. *Journal of the Association for Computing Machinery*, 29:841–862, July 1982.
- [5] Yulia Babovich. Cmodels, a system computing answer sets for tight logic programs, 2002.
- [6] Sabrina Baselice, Piero A. Bonatti, and Giovanni Criscuolo. On finitely recursive programs. In *Proceedings of the 23rd International Conference Logic Programming (ICLP'07)*, pages 89–103, 2007.
- [7] Sabrina Baselice, Piero A. Bonatti, and Michael Gelfond. Towards an integration of answer set and constraint solving. In *Proceedings of the 21st International Conference Logic Programming (ICLP'05)*, pages 52–66. Springer-Verlag, 2005.
- [8] R. J. Jr. Bayardo and R.C Schrag. Using CSP look-back techniques to solve real world sat instances. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 203–208, 1997.
- [9] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.
- [10] Piero A. Bonatti. Finitary open logic program. In *Proceedings of the 2nd International Workshop on Advances in Theory and Implementation in Answer Set Programming (ASP'03)*, 2003.
- [11] Piero A. Bonatti. Reasoning with infinite stable models. *Artificial Intelligence*, 156(1):75–111, 2004.
- [12] Gerhard Brewka, Ilkka Niemelä, and Tommi Syrjänen. Implementing ordered disjunction using answer set solvers for normal programs. In *The Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, 2002.

- [13] Gerhard Brewka, Ilkka Niemelä, and Tommi Syrjänen. Logic programs with ordered disjunction. *Computational Intelligence*, 20(2):333–357, 2004.
- [14] Francesco Buccafurri, Wolfgang Faber, and Nicola Leone. Disjunctive logic programs with inheritance. *Theory and Practice of Logic Programming*, 2(3):293–321, 2002.
- [15] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and weak constraints in disjunctive datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 2–17. Springer-Verlag, 1997.
- [16] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *Knowledge and Data Engineering*, 12(5):845–860, 2000.
- [17] Hans Kleine Büning, Marek Karpinski, and Andreas Flögel. Resolution for quantified boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
- [18] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.
- [19] Francesco Calimeri and Giovambattista Ianni. External sources of computation for answer set solvers. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’05)*, pages 105–118, 2005.
- [20] Francesco Calimeri and Giovambattista Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Communications*, 19(3):193–206, 2006.
- [21] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and computational properties of logic programs with aggregates. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI’05)*, pages 406–411, 2005.
- [22] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. C-plan: A conformant planner based on satisfiability. In *IJCAI 2001 Workshop on Planning under Uncertainty and Incomplete Information*, pages 98–103, 2001.
- [23] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. Sat-based planning in complex domains: concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1-2):85–117, 2003.
- [24] Luis Castro, Terrance Swift, and David Scott Warren. *The XSB System Version 3.0 Volume 2: Libraries, Interfaces and Packages*. chapter 12. Available at: <http://xsb.sourceforge.net/manual2/node166.html>.

- [25] Douglas Cenzer, Jeffrey B. Remmel, and Victor W. Marek. Logic programming with infinite sets. *Annals of Mathematics and Artificial Intelligence*, 44:309–339, 2005.
- [26] Ashok K. Chandra and David Harel. Computable queries for relational database systems. *Journal of Computer and System Sciences*, 21:156–178, 1980.
- [27] Ashok K. Chandra and David Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 2:1–15, 1985.
- [28] David Chapman. Planning for conjunctive goals. Technical report, Massachusetts Institute of Technology, 1985.
- [29] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 331–337, 1991.
- [30] Weidong Chen and David Scott Warren. Tabled evaluation with delaying for general logic programs. *Journal of the Association for Computing Machinery*, 43:20–74, 1 1996.
- [31] Paweł Cholewiński, Victor W. Marek, and Mirosław Truszczyński. Default reasoning system DeReS. In Luigia Carlucci Aiello, Jon Doyle, and Stuart Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kaufmann, San Francisco, California, 1996.
- [32] Keith L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
- [33] Alain Colmerauer, Henry Kanoui, Pasero Robert, and Roussel Philippe. Un système de communication en français, preliminary report. Technical report, October 1972.
- [34] Stephen A. Cook and Robert A Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.
- [35] Chiara Cumbo, Wolfgang Faber, and Gianluigi Greco. Improving query optimization for disjunctive datalog. In *Proceedings of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, pages 252–262, 2003.
- [36] Chiara Cumbo, Wolfgang Faber, and Gianluigi Greco. Enhancing the magic-set method for disjunctive datalog programs. In *Proceedings of the the 20th International Conference on Logic Programming (ICLP'04)*, pages 371–385, 2004.
- [37] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming.

In *Proceedings of the Twelfth Annual IEEE Conference on Computational Complexity*, pages 82–101, Ulm, Germany, June 1997. IEEE Computer Society Press.

- [38] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374–425, 2001.
- [39] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
- [40] Martin D. Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, complexity, and languages (2nd ed.): fundamentals of theoretical computer science*. Academic Press Professional, Inc., San Diego, CA, USA, 1994.
- [41] James P. Delgrande, Torsten Schaub, and Hans Tompits. Logic programs with compiled preferences. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, pages 464–468, Berlin, Germany, August 2000.
- [42] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.
- [43] Marc Denecker, Nikolay Pelov, and Maurice Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proceedings of the 17th International Conference on Logic Programming (ICLP'01)*, pages 212–226, 2001.
- [44] Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, 9(2):1–52, 2008.
- [45] Bistra N. Dilkina, Carla P. Gomes, and Ashish Sabharwal. Trade-offs in the complexity of backdoor detection. In *Principles and Practice of Constraint Programming (CP'07), 13th International Conference*, pages 256–270, 2007.
- [46] Yannis Dimopoulos. On computing logic programs. *Journal of Automated Reasoning*, 17:259–289, 1996.
- [47] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181, 1997.
- [48] Jürgen Dix, Ugur Kuter, and Dana Nau. Htn planning in answer set programming. Technical Report CS-TR-4336 (UMIACS-TR-2002-14), Dept. of CS, University of Maryland, College Park, MD 20742, February 2002.

- [49] Semra Doğandağ, F. Nur Alpaslan, and Varol Akman. Using stable model semantics (smodels) in the causal calculator (ccalc). In *The Tenth Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN'01)*. 2001.
- [50] Semra Doğandağ, Paolo Ferraris, and Vladimir Lifschitz. Almost definite causal theories. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04)*, pages 74–86. Springer-Verlag, 2004.
- [51] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.
- [52] Deborah East and Mirosław Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of KI 2001: Advances in Artificial Intelligence*, pages 138–153, 2001.
- [53] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. The diagnosis frontend of the dl原因 system. *AI Communications*, 12(1-2):99–111, 1999.
- [54] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Planning under incomplete knowledge. In *Proceedings of the First International Conference on Computational Logic (CL 2000)*, pages 807–821, 2000.
- [55] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *Journal of Artificial Intelligence Research (JAIR)*, 19:25–71, 2003.
- [56] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, ii: The DLVK system. *Artificial Intelligence*, 144:157–211, March 2003.
- [57] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. *ACM Transactions on Computational Logic*, 5(2):206–263, 2004.
- [58] Thomas Eiter, Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Computing preferred answer sets by meta-interpretation in answer set programming. Research Report 1843–02–01, Institut Für Informationssysteme, Technische Universität Wien, January 2002.
- [59] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of the 19th International Conference on Logic Programming*, pages 224–238, 2003.

- [60] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.
- [61] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. dlhex: A system for integrating multiple semantics in an answer-set programming framework. In *WLP*, pages 206–210, 2006.
- [62] Thomas Eiter, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. In *Proceedings of the 9th International Conference on the Principles of Knowledge Representation and Reasoning (KR’04)*, pages 141–151, 2004.
- [63] Thomas Eiter, Hans Tompits, and Stefan Woltran. On solution correspondences in answer set programming. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 96–102, 2005.
- [64] Charles Elkanan. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.
- [65] Wolfgang Faber. Decomposition of nonmonotone aggregates in answer set programming. In *20th Workshop on Logic Programming (WLP’06)*, pages 164–171, 2006.
- [66] Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets and their application to data integration. *Journal of Computer and System Sciences*, 73(4):584–609, 2007.
- [67] Wolfgang Faber and Nicola Leone. On the complexity of answer set programming with aggregates. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’07)*, pages 97–109, 2007.
- [68] Wolfgang Faber, Nicola Leone, Cristinel Mateis, and Gerald Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming*, 1999.
- [69] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA’04)*, pages 200–212, 2004.
- [70] François Fages. Consistency of Clark’s completion and existence of stable models. *Methods of Logic in Computer Science*, 1:51–60, 1994.
- [71] Paolo Ferraris and Enrico Giunchiglia. Planning as satisfiability in nondeterministic domains. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 748–753, 2000.

- [72] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.
- [73] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [74] Raphael A. Finkel, V. Wiktor Marek, and Mirosław Truszczyński. Constraint lingo: A program for solving logic puzzles and other tabular constraint problems. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 513–516, 2002.
- [75] Melvin Fitting and Marion Ben-Jacob. Stratified and three-valued logic programming semantics. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1054–1069, Seattle, 1988. The MIT Press.
- [76] Haim Gaifman, Harry Mairson, Yehoshua Sagiv, and Moshe Y. Vardi. Undecidable optimization problems for database logic programs. *Journal of the ACM*, 40(3):683–713, 1993.
- [77] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *Computing Surveys*, 16, June 1984.
- [78] Michael R. Garey and David S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman and Co, 1979.
- [79] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, 2007.
- [80] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007.
- [81] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A new grounder for answer set programming. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, pages 266–271, 2007.
- [82] Michael Gelfond. The USA-Advisor: A case study in answer set programming. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, pages 566–568, London, UK, 2002. Springer-Verlag.

- [83] Michael Gelfond and Nicola Leone. Logic programming and knowledge representation — the A-Prolog perspective. *Artificial Intelligence*, 138:3–38, 2002.
- [84] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. The MIT Press, August 1988.
- [85] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In *Proceedings of the 7th International Conference on Logic Programming*, pages 579–597, Jerusalem, Israel, June 1990. The MIT Press.
- [86] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
- [87] Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.
- [88] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.
- [89] Michael Gelfond and T. Son. Reasoning with prioritized defaults. In *Selected Papers presented at the Workshop on Logic Programming and Knowledge Representation (LPKR'97)*, pages 164–223, 1998.
- [90] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109, 1994.
- [91] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Causal laws and multi-valued fluents. In *Proceedings of Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)*, 2001.
- [92] Enrico Giunchiglia, G. Neelakantan Kartha, and Vladimir Lifschitz. Representing action: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–438, September 1997.
- [93] Carla P. Gomes, Bart Selman, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24:67–100, 2000.
- [94] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 240–254. Springer-Verlag, 1999.

- [95] Keijo Heljanko and Ilkka Niemelä. Answer set programming and bounded model checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*. AAAI, 2001.
- [96] Keijo Heljanko and Ilkka Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3:519–550, 2003.
- [97] Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. Undecidable boundedness problems for datalog programs. *Journal of Logic Programming*, 25:163–190, 1995.
- [98] David A. Huffman. Impossible objects as nonsense sentences. *Machine Intelligence*, 6:295–323, 1971.
- [99] Katsumi Inoue and Chiaki Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35:39–78, 1998.
- [100] Tomi Janhunen. Comparing the expressive powers of some syntactically restricted classes of logic programs. In *Proceedings of the First International Conference on Computational Logic (CL 2000)*, pages 852–866, London, UK, July 2002.
- [101] Tomi Janhunen. Representing normal programs with clauses. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI’06)*, pages 358–362, 2004.
- [102] Tomi Janhunen, Ilkka Niemelä, Patrik Simons, and Jia-Huai You. Unfolding partiality and disjunctions in stable model semantics. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference*, pages 411–419. Morgan Kaufmann Publishers, April 2000.
- [103] Tomi Janhunen and Emilia Oikarinen. Capturing parallel circumscription with disjunctive logic programs. In *The Proceedings of the 9th International Conference on Logics in Artificial Intelligence (JELIA’04)*, pages 134–146, 2004.
- [104] Dana Nau Juergen Dix, Ugur Kuter. Planning in answer set programming using ordered task decomposition. In B. Neumann A.Günther, R. Kruse, editor, *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI ’03), Hamburg, Germany*, LNAI 2821, pages 490–504, Berlin, 2003. Springer.
- [105] Matti Jarvisalo, Tommi A. Junttila, and Ilkka Niemelä. Unrestricted vs restricted cut in a tableau method for boolean circuits. *Annals of Mathematical Artificial Intelligence*, 44:373–399, 2005.
- [106] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence*, 1999.

- [107] David B. Kemp and Peter J Stuckey. Semantics of logic programs with aggregates. In *Proceedings of the International Logic Programming Symposium*, pages 387–401, 1991.
- [108] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th innovative applications of artificial intelligence conference*, pages 1368–1373. AAAI Press, 2005.
- [109] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 274–289, Bonn, Germany, September 1996. The MIT Press.
- [110] Nicola Leone, Simona Perri, and Francesco Scarcello. Improving asp instantiators by join-ordering methods. In *Proceedings of the 6th International Conference Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, Vienna, Austria, September 2001.
- [111] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2006.
- [112] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation (2nd ed)*. Prentice Hall, 1998.
- [113] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.
- [114] Vladimir Lifschitz. Answer set planning. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*, pages 25–37, Las Cruces, New Mexico, December 1999. The MIT Press.
- [115] Vladimir Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2001.
- [116] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
- [117] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37, 1994.
- [118] Vladimir Lifschitz and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:25–369, 1999.

- [119] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of the 18th National Conference on Artificial Intelligence*, pages 112–118, Edmonton, Alberta, Canada, July/August 2002. The AAAI Press.
- [120] Zhijun Lin, Yuanlin Zhang, and Hector Hernandez. Fast SAT-based answer set solver. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*. AAAI Press, 2006.
- [121] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
- [122] Lionello Lombardi. Mathematical structure of nonarithmetic data processing procedures. *Journal of the Association of Computing Machinery (JACM)*, 9(1):136–159, 1962.
- [123] Alan K. Machworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. John Wiley and Sons, 1987.
- [124] Tomi Männistö, Timo Soinen, Juha Tiuhonen, and Reijo Sulonen. Framework and conceptual model for reconfiguration. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.
- [125] Rainer Manthey and François Bry. SATCHMO: a theorem prover implemented in prolog. In *Proceedings of the 9th International Conference on Automated Deduction (CADE-88)*, pages 415–434, 1988.
- [126] Victor W. Marek, Ilkka Niemelä, and Mirosław Truszczyński. Logic programs with monotone abstract constraint atoms. *Theory and Practice of Logic Programming*, 8:167–199, 2008.
- [127] Victor W. Marek and Jeffrey B. Remmel. On the foundations of answer set programming. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 124–131. AAAI Press, March 2001.
- [128] Victor W. Marek and Jeffrey B. Remmel. On logic programs with cardinality constraints. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 219–228, 2002.
- [129] Victor W. Marek and Jeffrey B. Remmel. Set constraints in logic programming. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LP-NMR'04)*, pages 154–167, 2004.
- [130] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. cs.LO/9809032.

- [131] Victor W. Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.
- [132] Victor W. Marek and Mirosław Truszczyński. *Nonmonotonic Logic*. Springer-Verlag, 1993.
- [133] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398. Springer-Verlag, 1999.
- [134] Viktor W. Marek, Anil Nerode, and Jeffrey B. Remmel. The stable models of a predicate logic program. *Journal of Logic Programming*, 21(3):129–153, 1994.
- [135] Norman McCain and Hudson Turner. Causal theories of action and change. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 460–465, Menlo Park, California, 1997. AAAI Press.
- [136] Norman McCain and Hudson Turner. Satisfiability planning with causal theories. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *ICPKR98*, pages 212–223. Morgan Kaufmann, San Francisco, California, 1998.
- [137] Drew McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21:35–55, Summer 2000.
- [138] Veena S. Mellarkod and Michael Gelfond. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS’08)*, pages 15–31, 2008.
- [139] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.
- [140] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artificial Intelligence*, 25:75–94, 1985.
- [141] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.
- [142] Jean-Marie Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica*, 18, December 1982.
- [143] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

- [144] Ilkka Niemela and Jussi Rintanen. On the impact of stratification on the complexity of nonmonotonic reasoning. In *ECAI Workshop on Knowledge Representation and Reasoning*, pages 275–295, 1992.
- [145] Ilkka Niemelä and Patrik Simons. Extending the smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.
- [146] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag.
- [147] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, December 1999.
- [148] Ilkka Niemelä, Patrik Simons, and Tommi Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, April 2000.
- [149] Sergei P. Odintsov and David Pearce. Routley semantics for answer sets. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’05)*, pages 343–355, 2005.
- [150] Emilia Oikarinen and Tomi Janhunen. Lpeq and dlpeq — translators for automated equivalence testing. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’04)*, pages 336–340, January 2004.
- [151] Emilia Oikarinen and Tomi Janhunen. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence*, 2006.
- [152] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc, 1994.
- [153] David Pearce and Agustín Valverde. Towards a first order equilibrium logic for nonmonotonic reasoning. In *The Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA’04)*, pages 147–160, 2004.
- [154] Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 7:301–354, May 2007.
- [155] Simona Perri and Nicola Leone. Parametric connectives in disjunctive logic programming. *AI Communications*, 17(2):63–74, 2004.

- [156] Axel Polleres. *Advances in Answer Set Planning*. PhD thesis, Institut für Informationssysteme, Technische Universität Wien, September 2003.
- [157] Hemantha Ponnuru, Raphael A. Finkel, Victor W. Marek, and Mirosław Truszczyński. Automatic generation of English-language steps in puzzle solving. In *Proceedings of the International Conference on Artificial Intelligence, (IC-AI'04)*, pages 437–442, 2004.
- [158] Teodor C. Przymusiński. Stationary semantics for disjunctive logic programs. In *Proceedings of the North American Logic Programming Conference*, pages 40–59, Austin, Texas, 1990. MIT Press.
- [159] Teodor C. Przymusiński. Stable semantics for disjunctive programs. *New Generation Computing Journal*, 9(3):401–424, 1991.
- [160] Teodor C. Przymusiński. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1989.
- [161] Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, and Juliana Freire. XSB: A system for efficiently computing WFS. In *Logic Programming and Non-monotonic Reasoning (LPNMR'97)*, pages 431–441, 1997.
- [162] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.
- [163] Francesco Ricca, Wolfgang Faber, and Nicola Leone. A backjumping technique for disjunctive logic programming. *AI Communications – The European Journal on Artificial Intelligence*, 19:155–172, 2006.
- [164] Francesco Ricca and Nicola Leone. Disjunctive logic programming with types and objects: The dlv^+ system. *Journal of Applied Logic*, 5(3):545–573, 2007.
- [165] Jussi Rintanen. Lexicographic priorities in default logic. *Artificial Intelligence*, 106(2):221–265, 1998.
- [166] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170:1031–1080, 2006.
- [167] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 1995.
- [168] Domenico Saccà and Carlo Zaniolo. Partial models and three-valued stable models in logic programs with negation. In *Proceedings of the Workshop on Nonmonotonic Reasoning and Logic Programming*, pages 87–101, 1991.

- [169] Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1–2):185–222, 2000.
- [170] Torsten Schaub and Kewen Wang. A comparative study of logic programs with preference. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI’01)*, pages 597–602, 2001.
- [171] John S. Schlipf. The expressive powers of logic programming semantics. *Journal of Computer and System Sciences*, 51(1):64–86, 1995.
- [172] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Publishing Company, Inc, 1990.
- [173] Josefina Sierra-Santibañez. Heuristic planning: a declarative approach based on strategies for action selection. *Artificial Intelligence*, 153(1–2):307–337, 2004.
- [174] Patrik Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.
- [175] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
- [176] Raymond Smullyan. *What is the name of this book?* Simon & Schuster, Inc., 1978.
- [177] Timo Soininen. *An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks*. PhD thesis, Helsinki University of Technology, Finland, 2000.
- [178] Timo Soininen and Niemelä Ilkka. Formalizing configuration knowledge using rules with choices. Technical Report TKO-B142, Laboratory of Information Processing Science, Helsinki University of Technology, 1998.
- [179] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*. Springer-Verlag, January 1999.
- [180] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge*, Stanford, USA, March 2001.
- [181] Tran Cao Son, Chitta Baral, Nam Tran, and Sheila McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic (TOCL)*, 7:613–657, 2006.

- [182] Tran Cao Son and Enrico Pontelli. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming*, 7:355–375, May 2007.
- [183] Tran Cao Son, Enrico Pontelli, and Phan Huy Tu. Answer sets for logic programs with arbitrary abstract constraint atoms. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*, pages 129–134, 2006.
- [184] Leon Sterling and Ehud. Shapiro. *The Art of Prolog*. MIT press, 1994.
- [185] Gerald Sussman. *A Computer Model of Skill Acquisition*. Elsevier Science Inc, New York, USA, 1975.
- [186] Tommi Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B 18, Helsinki University of Technology, Helsinki, Finland, October 1998.
- [187] Tommi Syrjänen. A rule-based formal model of software configuration. Research Report A 55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.
- [188] Tommi Syrjänen. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic*, London, UK, July 2000. Springer-Verlag.
- [189] Tommi Syrjänen. Modelling the game of life using logic programs. In Nisse Husberg, Tomi Janhunen, and Ilkka Niemelä, editors, *Leksa Notes in Computer Science, Festschrift in Honour of Professor Leo Ojala*, pages 115–124. 2000.
- [190] Tommi Syrjänen. Optimizing configurations. In *Proceedings of the ECAI Workshop W02 on Configuration*, pages 85–90, Berlin, Germany, August 2000.
- [191] Tommi Syrjänen. Omega-restricted logic programs. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR’01)*, Vienna, Austria, September 2001. Springer-Verlag.
- [192] Tommi Syrjänen. Version spaces and rule-based configuration management. In *Proceedings of the IJCAI Workshop on Configuration*, August 2001.
- [193] Tommi Syrjänen. Logic programming with cardinality constraints. Research Report A 86, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 2003.

- [194] Tommi Syrjänen. Cardinality constraint logic programs. In *The Proceedings of the 9th European Conference on Logics in Artificial Intelligence (JELIA '04)*, pages 187–200, Lisbon, Portugal, September 2004. Springer-Verlag.
- [195] Tommi Syrjänen. Debugging inconsistent answer set programs. In *Proceedings of the 11th International Workshop on Non-Monotonic Reasoning*, Lake District, UK, May 2006.
- [196] Tommi Syrjänen and Ilkka Niemelä. The Smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 77–84, Vienna, Austria, September 2001. Springer-Verlag.
- [197] Tommi Syrjänen. *Lparse User's Manual*. Available at: <http://www.tcs.hut.fi/Software/smodels>.
- [198] Tommi Syrjänen. Answer set programming and 52 graph problems (manuscript), 2008.
- [199] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1936.
- [200] Alan Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228, 1939.
- [201] Hudson Turner. Polynomial-length planning spans the polynomial hierarchy. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, pages 111–124, 2002.
- [202] Jeffrey D. Ullman. *Principles of database and knowledge-base systems Vol 1*. Rockville, 1989.
- [203] Jeffrey D. Ullman. *Principles of database and knowledge-base systems Vol 2*. Rockville, 1989.
- [204] Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.
- [205] Allen van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, 6:109–133, 1989.
- [206] Allen van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47:185–221, August 1993.
- [207] Allen van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, July 1991.

- [208] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20:93–123, Summer 1999.
- [209] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.
- [210] Terry Winograd. *Understanding Natural Language*. Academic Press, 1972.
- [211] Johan Wittocx, Joost Vennekens, Maarten Mariën, Marc Denecker, and Maurice Bruynooghe. Predicate introduction under stable and well-founded semantics. In *Proceedings of the 22nd International Conference Logic Programming (ICLP'06)*, pages 242–256. Springer-Verlag, 2006.
- [212] Stefan Woltran. Characterizations for relativized notions of equivalence in answer set programming. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence*, pages 161–173, 2004.
- [213] Jia-Huai You and Li-Yan Yuan. A three-valued semantics for deductive database and logic programs. *Journal of Computer and System Science*, 49:334–361, 1994.
- [214] Jia-Huai You and Li-Yan Yuan. On the equivalence of semantics for normal logic programs. *Journal of Logic Programming*, 20(1):79–89, 1995.
- [215] Yan Zhang and Norman Y. Foo. Answer sets for prioritized logic programs. In *International Logic Programming Symposium*, pages 69–83, 1997.

Index

- \perp (an atom that is always false), 13
- \perp (an atom that is always false), 15
- \models (a satisfaction relation), 19
- \models_I (an I -satisfaction relation), 36
- \top (an atom that is always true), 13
- \uparrow (the composition operator), 23
- 2-**EXP** complexity class, 76
- 2-**NEXP** complexity class, 76

- A (an arbitrary atom), 13
- \mathcal{A} (action names), 175
- ACCP (augmented cardinality constraint programs), 13
- ACCP (augmented programs), 133
- $\text{act}(A, t)$ (planning action translation), 184
- action language, 173
- action description, 175
- action language
 - causal theory, 189
- action language \mathcal{B} , 175
- action signature, 175
- action translation (planning), 184
- aggregate
 - literal, 115
- aggregate
 - count, 72
- aggregates
 - relational, 67
- application, 30
- approximated size, 88
- arguments, 13
- atom, 13
 - cardinality, 14, 118, 119, 121
 - simple, 14
 - cardinality bound, 14
 - \mathcal{I} (planning atom), 199
 - justification, 27
- atomic term, 12
- $\text{Atoms}(P)$ (atoms of P), 15
- $\text{Atoms}_{\mathbf{H}}(P)$ (the Herbrand base of P), 29
- attribute name, 66
- augmented basic programs, 133
- augmented standard interpretation, 127
- \mathcal{B} (action language), 175
- back end, 133
- backdoor, 104
- backjumping, 146
- base, 29
 - Herbrand, 29
- basic cardinality constraint program, 15
- basic language, 12
- basic literal, 13
- basic rule, 14
- binder, 155
- binding, 135
- bind-variables*, 137
- BLOCKS WORLD, 176
- body, 14
- $\text{body}_{\mathcal{L}}^-(R)$ (the negative conditional literals in R), 16
- $\text{body}^+(R)$ (the positive basic literals in R), 16
- $\text{body}_{\mathcal{L}}^+(R)$ (the positive conditional literals in R), 16
- BOOLEAN SAT, 161
- bound, 14
- BOUNDED PLANNING, 181
- \mathcal{B}_P (planning action language), 198
- brave consequence, 33

- \mathcal{C} (a cardinality literal), 14
- can-bind*, 137
- cardinality atom
 - reduct, 20
- cardinality atom, 14, 118, 119
 - bound, 14
- cardinality literal, 14
- causal theory, 189
- cautious consequence, 33

- CCP, 15
- CCP (cardinality constraint program), 13
- CCP-solver, 101
 - partial, 102
- choice point, 101
- choice rule, 14
- classical model, 19
- classical negation, 129
- Cn (planning consequences), 177
- completeness, 76
- complexity class
 - polynomial time, 75
- complexity results, 91
- compound term, 12
- computation tree, 101
- compute-extensions*, 151
- $cond(\mathcal{L})$ (planning condition), 175
- $cond(\mathcal{L})$ (the condition of \mathcal{L}), 14
- condition, 13
- condition (planning), 175
- conditional effect (planning), 181
- conditional literal, 13, 122
- consequence, 33
- constant, 12
- count aggregate, 72
- create-stratification*, 53

- $body_D(R)$ (the domain literals of R), 49
- $\mathcal{D}(P)$ (the domain predicates of P), 49
- $\mathfrak{D}(P)$ (the dependency relation of P), 47
- \mathcal{D} (action description), 198
- \mathcal{D} (action description), 175
- $\mathfrak{D}_1(P)$ (the one-step dependency relation of P), 46
- data predicate, 159
- Datalog, 68
- Datalog[−], 68
- decision problem, 75
- dependency relation
 - negative, 47
 - one-step, 46
- dependency path, 46
- dependency relation, 47
- derivation length, 23
- $len(A, P)$ (the derivation length of A in P), 23
- $\mathfrak{D}^-(P)$ (the negative dependency relation of P), 47
- $domain(\mathcal{I})$ (planning domain translation), 204
- domain translation (planning), 204
- domain computation
 - as constraint satisfaction, 153
- domain computation, 148
 - database method, 152
 - discarded negations method, 152
- domain computer, 135
- domain literal, 49
- domain model, 62
- domain predicate, 49
- domain program, 56
- dynamic law, 175

- $\mathcal{E}(L_c, U)$ (the expansion of L_c), 30
- E_A (planning conditional effects), 181
- ECCP (extended cardinality constraint programs), 13
- $eff(\mathcal{L})$ (planning effect), 175
- effect (planning), 175
- elementary actions, 175
- encoding
 - BOOLEAN SAT, 162
 - MAXSAT, 163
 - planning translation, 182
 - SAT, 159
 - Turing machine, 79
 - uniform, 108, 158
 - VERTEX COLORING, 158
- equivalence, 32
- eval-rule*, 68
- EXP** complexity class, 76
- expand*, 151
- expansion, 30
 - relevant, 60
- expression, 17
 - correctly typed (planning), 200
 - ground, 17
- extended rule, 118
- extension, 32
- extensional predicate, 68

- \mathcal{F} (fluents), 175
- fact, 14

FCCP (full cardinality constraint pro- I -Instantiation, 36
 gram), 13
find-plan, 206
 finitary programs, 117
 fluent
 default value, 208
 fixed, 208
 free, 208
 inertia, 183, 203, 208
 totalizing, 184, 202
 fluent inertia (planning), 183, 203
 fluents, 175
 $F(M)$, 56
 formula, 17
 front end, 133
 full language, 118
 function
 interpretation, 34
 interpreted, 33
 function symbol, 12
 function problem, 75
 function term, 12
 FUNCTIONAL MODEL, 101

 generate and test, 157
 generator, 157
get-next-instance, 137
 global variable, 17
 $\text{goal}(S_g)$ (planning goal translation),
 204
 ground term, 12
 ground expression, 17

 hardness, 76
has-stable-model, 64
 $\langle H, B \rangle$ (a rule $H \leftarrow B$), 16
 head, 14
 $\text{head}(R)$ (the head of R), 16
 Herbrand
 base, 29
 instantiation, 31
 relevant instantiation, 61
 universe, 29
 $\text{inst}_{\mathbf{Hr}}(P, M)$ (the relevant instanti-
 ation of P), 61

 I_s (the standard interpretation), 41
 \mathcal{I} (planning instance), 176
 \mathcal{I} (planning structure), 199
 I -satisfaction, 36

 implementation
 overview, 133
 induced LTS, 177
 $\text{inertia}(F, t)$ (inertia translation), 183
 $\text{inertia}(p, t)$ (inertia translation), 203
 $\text{initial}(S_i)$ (planning initial transla-
 tion), 204
instantiate-local, 137
instantiate-relevant, 139
instantiate-rule, 135, 137, 139
 instantiation, 30
 Herbrand, 31
 interpreted, 36
 planning, 200
 relevant, 55, 61
 relevant Herbrand, 61
 rule, 30
instantiation, 89
 instantiator, 135
 integral range, 124
 intensional predicate, 68
 interpretation, 34
 arithmetic, 41
 augmented standard, 127
 numbers, 41
 numerically interpreted constants,
 42
 standard, 41
 interpreted function, 33
 iterator, 137

 kakuro puzzles, 168
 Knaster-Tarski -operator, 22

 L (an arbitrary basic literal), 13
 \mathbf{L} (labeled transition system), 177
 \overline{L} (the complement of a literal L),
 13
 \mathcal{L}_d (planning dynamic laws), 198
 \mathcal{L}_d (planning dynamic laws), 175
 \mathcal{L}_s (planning static laws), 175, 198
 labeled transition system, 177
 induced, 177
 language
 action, 173
 causal theory, 189
 language hierarchy, 132
 law
 dynamic, 175

- static, 175
- law(S, t) (planning static law translation), 185
- law(S, t) (planning static law translation), 202
- least model, 19
- LINEARIZATION, 192
- linearization (planning), 192
- $lit(\mathcal{L})$ (the main literal of \mathcal{L}), 14
- literal
 - aggregate, 115
 - basic, 13
 - cardinality, 14
 - closed set, 177
 - complete set, 175
 - conditional, 13, 122
 - conditional expansion, 30
 - conditional in rule body, 123
 - domain, 49
 - main, 13
 - negative, 13
 - positive, 13, 15
 - weight, 127
- local variable, 13
- logical consequence, 33
- LTS (labeled transition system), 177
- $\mathbf{M}(P)$ (all classical models of P), 19
- main literal, 13
- MAXSAT, 162
- minimal model, 19
- $\mathbf{MM}(P)$ (the minimal models of P), 19
- model, 19
 - classical, 19
 - domain, 62
 - function problem, 101
 - I -stable, 37
 - least, 19
 - minimal, 19
 - partial domain, 62
 - stable, 25, 32
 - supported, 27
- modularity, 108, 158
 - of transformation, 108
- naive-datalog*, 68
- naive-instantiate-relevant*, 64
- natural join, 67
- $\text{neg}(C)$ (negative conditional literals in C), 14
- negation
 - classical, 129
 - strong, 129
- negative
 - literal, 13
- negative dependency, 47
- NEXP** complexity class, 76
- no-op, 180
- nondeterminism, 76
- NP** complexity class, 76
- numerals, 81
- $\Omega(R, \mathcal{S})$ (the ω -valuation of R), 50
- ω -restriction, 50
- ω -stratification, 47
- ω -valuation, 50
- one-step dependency relation, 46
- optimization, 167
- oracle, 163
 - basic form, 163
 - solving function problems, 164
 - two program construction, 165
 - with exclusion, 164
- P** (planning atoms), 177
- $\mathcal{P}(P)$ (the program predicates of P), 49
- P_A (planning precondition), 181
- parametric connectives, 113
- parser, 134
- partial CCP-solver, 102
- partial domain model, 62
- partial domain program, 56
- $P_C(t)$, 72
- $P_{\mathcal{D}}$ (the domain program), 56
- PDDL, 174
- π_P (a dependency path), 46
- P_k (the k th stratum program), 56
- $P_{\leq k}$ (the k th partial domain program), 56
- plan, 178
 - parallel, 178
 - proper, 180
 - sequential, 178
- planning
 - action translation, 184
 - BOUNDED PLANNING, 181
 - condition, 175

- conditional effect, 181
- different forms, 174
- effect, 175
- executability, 210
- fluent inertia, 183, 203
- inertia translation, 183, 203
- instance, 199
- instantiation, 200
- linearization, 192
- no-op, 180
- parallel, 191
- plan generation, 206
- precondition, 181, 203, 209
- predicate signature, 197
- static law translation, 185, 202
- temporal, 210
- translation, 202
- variables, 197
- variants, 207
- PLANNING, 181
- planning problem, 176
- P^M (reduct of P w.r.t. M), 20
- P_ω (the non-domain program), 58
- $\text{pos}(C)$ (positive conditional literals in C), 14
- positive
 - literal, 13, 15
 - rule, 14
- $P \mid p$ (the rules for predicate symbol p), 15
- precondition, 209
- precondition (planning), 181
- precondition (planning), 203
- $\text{Preds}(P)$ (predicate symbols occurring in P), 15
- $\text{pred}(A)$ (predicate symbol of A), 13
- predicate
 - data, 68, 159
 - domain, 49
 - extension, 32
 - extensional, 68
 - intensional, 68
 - program, 68, 159
 - signature (planning), 197
 - standard, 42
 - type (planning), 201
- predicate symbol, 13
- problem
 - completeness, 76
 - decision, 75
 - function, 75
 - hardness, 76
 - INSTANTIATION, 74
 - MODEL, 74
 - planning \mathcal{B} , 176
- program
 - cardinality constraint, 15
 - consequence, 33
 - disjunctive, 191
 - domain, 56
 - equivalence, 32
 - finitary, 117
 - full language, 118
 - instantiation, 31
 - λ -restricted, 154
 - normal, 18
 - normal as a CCP, 28
 - ω -restricted, 51
 - optimization, 167
 - partial domain, 56
 - proper, 15
 - simple, 15
 - size, 87
 - stratified, 58
 - stratum, 56
 - top, 57
- program predicate, 159
- programming methodology, 157
- projection, 66
- proper
 - plan, 180
 - program, 15
- provability operator, 22
- PTIME**, 75
- $P(U, M)$ (the top program), 57
- 2-QBF (quantified boolean formula), 165
- R** (LTS transition relation), 177
- range, 118
- range-restriction, 43
- \mathbf{R}_C , 72
- reduct, 20
 - cardinality atom, 20
 - CCP, 20
 - conditional literal, 20
 - general concept, 18
 - normal program, 18

- rule, 20
 - weight literal, 128
- reduction, 75
- relational database model, 66
- relevant
 - expansion, 60
- relevant Herbrand instantiation, 61
- relevant expansion, 60
- relevant instantiation, 55, 61
- remove-binding*, 137
- renaming, 66
- reset-instances*, 137
- restriction
 - ω , 50
- $\mathbf{R}_{\mathcal{L}}$, 72
- \mathbf{R}_p , 71
- \mathbf{R}_R , 72
- $\mathbf{R}_{R,\mathcal{D}}$, 71
- rule
 - basic, 14
 - body, 14
 - choice, 14
 - empty head, 15
 - extended, 118
 - fact, 14
 - head, 14
 - I*-Instantiation, 36
 - instantiation, 30
 - pair notation, 16
 - positive, 14
 - reduct, 20
 - rewriting, 145
 - smodels, 155
 - smodels basic rule, 155
 - smodels choice rule, 156
 - smodels constraint rule, 156
 - smodels weight rule, 156
 - translating to smodels, 156
- \mathcal{S} (planning action signature), 175
- \mathcal{S} (planning predicate signature), 197
- \mathbf{S} (LTS states), 177
- \mathcal{S} (an ω -stratification), 47
- \mathcal{S} (action signature), 175
- σ , 30
- SAT, 159
- SAT, 104
- satisfaction, 19
 - of planning instance, 178
- SCC graph, 52
- SCCP (simple cardinality constraint programs), 13
- search tree, 102
- selection, 66
- signature
 - action, 175
- simple
 - cardinality atom, 14
- simple program, 15
- size
 - approximated, 88
 - explosion, 88
 - of a program, 87
- $\text{size}(P)$ (of a program), 87
- $\text{size}_A(P)$ (approximated size), 88
- Sokoban, 216
- SOKOBAN, 216
- solution, 102
- solver
 - CCP, 101
 - partial, 102
- splitting set, 57
- stable model
 - interpreted, 37
- stable model
 - of a ground CCP, 25
 - of a non-ground CCP, 32
- standard interpretation, 41
- standard predicate symbols, 42
- static law, 175
- stratification, 58
 - ω , 47
 - strict ω , 47
- stratified program, 58
- stratum program, 56
- strict ω -stratification, 47
- STRIPS, 174
- strong negation, 129
- $\text{subs}(V, U)$ (the set of all substitutions $V \rightarrow U$), 30
- substitution, 30
 - application, 30
 - \mathcal{B}_P (planning), 200
- supported models, 27
- symbol
 - function, 12
 - predicate, 13
- symbolic sets, 113

- syntax
 - basic, 12
 - full, 118
- $T(V, E)$ (planning type assignment), 200
- T_P (the provability operator), 22
- \mathbf{T} (LTS transitions), 177
- temporal planning, 210
- term, 12
 - atomic, 12
 - compound, 12
 - constant, 12
 - function, 12
 - ground, 12
 - integral range, 124
 - range, 118
 - variable, 12
- tester, 157
- $\text{time}(i)$ (planning time literal), 201
- tl (planning translation), 183
- $\text{tl}(\mathcal{D}, t)$ (complete planning translation), 185
- $\text{total}(F, t)$ (planning totalizing translation), 184, 202
- totalizing fluents (planning), 184, 202
- transformer, 134
- translation
 - domain (planning), 204
 - goal (planning), 204
 - initial (planning), 204
- tree
 - complete, 101
 - computation, 101
 - search, 102
- Turing machine
 - nondeterministic, 76
 - time-bounded, 75
- Turing machine
 - acceptance, 78
 - computation, 78
 - configuration, 78
 - definition, 78
 - determinizing, 101
 - encoding, 79
 - nondeterminist simulation, 86
 - nondeterministic acceptance, 79
 - nondeterministic definition, 79
 - numeral representation, 81
 - simulation, 83
 - space-bounded, 75
 - termination, 78
 - time-bounded, 78
- $\text{type}(E)$ (planning type literals), 202
- type assignment (planning), 200
- type identifier, 197
- type predicate (planning), 201
- $U(P)$ (the standard universe of P), 40
- $U_{\mathbf{H}}(P)$ (the Herbrand universe of P), 29
- uniform encoding, 108
- uniform encodings, 158
- universe, 29
 - Herbrand, 29
 - standard, 40
- $U_{\mathbf{H}}^s(P)$ (the simple Herbrand universe of P), 29
- \mathbf{V} (LTS valuation), 177
- \bar{V} , 119
- VACUUM WORLD, 211
- valuation
 - ω , 50
- $\text{Var}(E)$ (variables occurring in E), 17
- variable, 12
 - backdoor, 104
 - binding, 135
 - first-level, 17
 - global, 17
 - local, 13
 - substitution, 30
 - type assignment (planning), 200
- $\text{Var}_t(E)$ (the first level variables of E), 17
- $\text{Var}_g(E)$ (the global variables of E), 17
- $\text{Var}_l(E)$ (the local variables of E), 17
- VERTEX COLORING, 157
- visit*, 151
- visit-components*, 150
- weight literal, 127

TKK DISSERTATIONS IN INFORMATION AND COMPUTER SCIENCE

- TKK-ICS-D2 Laur, Sven
Cryptographic Protocol Design. 2008.
- TKK-ICS-D3 Harva, Markus
Algorithms for Approximate Bayesian Inference with Applications to Astronomical Data Analysis. 2008.
- TKK-ICS-D4 Ukkonen, Antti
Algorithms for Finding Orders and Analyzing Sets of Chains. 2008.
- TKK-ICS-D5 Tatti, Nikolaj
Advances in Mining Binary Data: Itemsets as Summaries. 2008.
- TKK-ICS-D6 Klami, Arto
Modeling of Mutual Dependencies. 2008.
- TKK-ICS-D7 Oikarinen, Emilia
Modularity in Answer Set Programs. 2008.
- TKK-ICS-D8 Salojärvi, Jarkko
Inferring Relevance from Eye Movements with Wrong Models. 2008.
- TKK-ICS-D9 Yang, Zhirong
Discriminative Learning with Application to Interactive Facial Image Retrieval. 2008.
- TKK-ICS-D10 Järvisalo, Matti
Structure-Based Satisfiability Checking: Analyzing and Harnessing the Potential. 2008.
- TKK-ICS-D11 Tikka, Jarkko
Input Variable Selection Methods for Construction of Interpretable Regression. 2008.

ISBN 978-951-22-9762-7 (Print)
ISBN 978-951-22-9763-4 (Online)
ISSN 1797-5050 (Print)
ISSN 1797-5069 (Online)