Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 86
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 86
Espoo 2003 HUT-TCS-A86

LOGIC PROGRAMMING WITH CARDINALITY CONSTRAINTS

Tommi Syrjänen

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

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 86
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 86
Espoo 2003 HUT-TCS-A86

LOGIC PROGRAMMING WITH CARDINALITY CONSTRAINTS

Tommi Syrjänen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-6896-5
ISSN 1457-7615

Multiprint Oy
Helsinki 2003

ABSTRACT: In this work we examine cardinality constraint logic programs. We give a formal definition of the stable model semantics for general cardinality constraint programs and define a syntactic subclass of them, omega-restricted programs, that stays decidable even when function symbols are used. We show that the computational complexity of omega-restricted programs is 2-NEXP-complete in the general case and NEXP-complete if function symbols are not used. We give a general framework for extending the semantics and give four extensions to the basic semantics, including classical negation and partial stable model semantics. We show how the extensions can be translated back to normal omega-restricted programs and give a similar translation for logic programs with ordered disjunction. We present some implementation details of omega-restricted programs and give several examples of their use.

KEYWORDS: Logic programming, cardinality constraint, instantiation, stable model semantics, smodels

CONTENTS

LIST OF SYMBOLS AND NOTATIONS

$\emptyset$	the empty set
$\cup$	set union
$\cap$	set intersection
$\wedge$	logical and
$\vee$	logical or
$\neg$	logical not (classical negation)
not	negation as failure
$\top$	an atom that is always true or the maximum truth value
$\bot$	the minimum truth value
$\rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m \rangle$	a literal set
$vars_{\mathfrak{L}}(S)$	local variables of S
$lit(S)$	the main literal of S
$conds(S)$	the conditions of S
$L \leq \{S_1, \ldots, S_n\} \leq U$	a cardinality constraint
$bound_L(C)$	the lower bound of C
$bound_U(C)$	the upper bound of C
$\mathcal{L}(C)$	the literal sets of C
$pred(A)$	the predicate symbol of the atom A
$vars(t)$	the set of variables occurring in t
$h \leftarrow l_1, \ldots, l_n$	a rule
$head(r)$	the head of r
$body^+(r)$	positive literals occuring in the body of r
$body^-(r)$	negative literals occuring in the body of r
$body_s(r)$	simple constraint literals occurring in r
P	a logic program
D	a data model
$\langle P, D \rangle$	a combined program and a data model
Σ	a vocabulary
U	an universe
I	an interpretation
$\mathcal{N}$	a set of names
$\mathcal{F}$	the set of all function symbols
$\mathcal{P}$	the set of all predicate symbols
$\mathcal{P}_D$	the set of data predicates
$\mathcal{P}_P$	the set of program predicates
$\uplus$	data model augmentation operator
D_P	the final data model of an ω-restricted program P
σ	a substitution
$inst(P, D)$	a partial instantiation of $\langle P, D \rangle$
$E(S, D)$	the expansion of S w.r.t. D
$E'(S, D)$	the simple expansion of S w.r.t. D
$E'_s(S, D)$	the satisfied expansion of S w.r.t. D
$\mathbf{HI}(P, D)$	the Herbrand instantiation of $\langle P, D \rangle$
$\mathbf{HI}_r(P, D)$	the relevant instantiation of $\langle P, D \rangle$
$\models$	a satisfaction relation

$cl(P)$	the deductive closure of P
P^M	reduct of P with respect to M
T_P	deductive closure operator
$\mathcal{A}(P, D)$	the set of all stable models of $\langle P, D\rangle$
$\mathfrak{D}$	a dependency relation
$\mathcal{S}$	an ω-stratification
Ω	an ω-valuation
P_i	a stratum program
$\mathfrak{C}$	a constraint literal signature
$\mathcal{T}$	a set of truth values
$\mathfrak{v}$	a valuator
$\mathfrak{s}$	a satisfier
$\mathfrak{I}(P, \mathcal{T})$	the set of all interpretations of P
$\mathfrak{r}$	a reducer
$\mathfrak{c}$	a constraint reducer
$\times$	an ordered disjunction
P	polynomial time complexity class
EXP	exponential time complexity class
2-**EXP**	doubly-exponential time complexity class
NC	nondeterministic complexity class C

1 INTRODUCTION

Logic programming (LP) is a way to solve problems using formal logic. The basic idea is to express the problem domain using a set of inference rules. Then, an inference engine is used to find the solution for the problem. In its purest form LP is a form of *declarative* programming where the programmer does not explicitly write the algorithms to solve the problems but instead describes what a valid answer should look like. At least this is how it works in theory. In practice, many LP systems have some nondeclarative aspects in them.

A normal logic program has four different kinds of syntactic elements: terms, atoms, literals, and rules. Terms denote the elements from the universe of the problem domain, and atoms express statements about the elements and their relationships. For example, the atom *parent(mother, son)* states that *mother* is a parent of *son*. A literal is either an atom A, or its negation $\mathrm{not}(A)$. In the classical case exactly one of A or $\mathrm{not}(A)$ is always true but there are semantics where this does not hold. A rule then has the form:

$$h \leftarrow l_1, \ldots, l_n \tag{1}$$

where an atom h is the rule *head* and the literals l_i form the rule *body*. The intuition is that if all literals l_i are true, then also h has to be true.

Numerous different semantics for logic programs have been proposed since van Emden and Kowalski published their ground breaking article *The Semantics of Predicate Logic as a Programming Language* [79] in 1976 where they presented the mathematical foundation of the Prolog programming language. It was soon found that Prolog had several weaknesses. For example, it is not fully declarative as changing the order of literals or rules may change the meaning of the program. We do not go into details here since they are out of the scope of this work. A detailed discussion on capabilities of Prolog can be found, for example, in Lloyd's *Foundations of Logic Programming* [37].

A number of different semantics have been proposed to overcome the problems of Prolog. One of them, the stable model semantics by Gelfond and Lifschitz [24, 25], has proven to be very suitable for many application domains such as product configuration [66] and planning [34, 33].

A stable model of a logic program P is a set of atoms that is a classical model[1] of P that fulfills a few additional conditions that will be formalized in Section 2. The stable models have two important properties:

1. *minimality*: If M is a stable model of P, then there is no stable model M' that is a strict subset of M.

2. *justification*: Each atom a true in M has a non-circular justification. That is, a occurs as a head of some rule with a satisfied body and where none of the body literals are derived using a itself.

[1] A set of atoms is a classical model of a program if it is a model of the set of clauses that is obtained by replacing each rule $h \leftarrow l_1, \ldots, l_n$ by the implication $l_1 \wedge \cdots \wedge l_n \rightarrow h$.

Consider a simple example:

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

Here we have only one stable model, namely $M_1 = \{a\}$. The other minimal classical model would be $M_2 = \{b, c\}$ but this is not stable since c is not justified in M_2.

The stable model semantics of normal logic programs is one form of the more general *Answer Set Programming* (ASP) paradigm [11, 21, 33, 39, 47]. In ASP a problem is encoded using a set of logical sentences P and the answer is then a set of atoms that is a model of P that usually has to satisfy the minimality and justification criteria. Most of the existing ASP systems are based on the stable model semantics for normal programs but a few are based on other formalisms. Among the systems are SMODELS [50], Assat [36], Cmodels [3], and NoMoRe [1] that work for normal programs, DLV for disjunctive programs, and ASPPS [18] for propositional schemata.

We can express a large number of problems nicely using normal logic programs and ASP, but sometimes we run into conditions that cannot be expressed compactly. For example, in product configuration we might have a rule of the form: "A computer must have at least one but at most four hard disk drives." A straightforward encoding of such a condition with n choices out of m possibilities needs at least $\binom{m}{n}$ rules to express it. However, constraints of this form can be easily expressed using the notion of cardinality constraint literals that are a generalization of constraint rules that were proposed in [68].

A cardinality constraint literal has the form:

$$L \leq \{l_1, \ldots, l_n\} \leq U \tag{2}$$

where U and L are integral lower and upper *bounds* and l_i are literals. The intuition is that (2) is true if the number of true literals l_i is between the bounds, inclusive. A constraint literal may occur in a rule in every place where a basic literal may. For example, the rule:

$$1 \leq \{\mathit{monitor}_{15"}, \mathit{monitor}_{17"}, \mathit{monitor}_{19"}\} \leq 1 \leftarrow \mathit{computer}$$

states that a computer has exactly one monitor that is of one of the three available types.

Most of the earlier work on cardinality constraints [51, 68, 50, 65] has concentrated on variable-free programs and variables have received only little attention. A rule with variables is seen as a schema denoting its all ground instances that are obtained by substituting constants that occur in the program for variables. This interpretation is intuitively clear and works well when the program does not have function symbols, but with functions the situation gets more complicated.

In some cases we would like to use functions for computation. For example, when we are instantiating the rule:

$$\mathit{rectangle\text{-}area}(X, Y, X \times Y) \leftarrow \mathit{number}(X), \mathit{number}(Y)$$

with variable bindings $\{X/3, Y/2\}$ we want to evaluate the product so that we get the ground rule:

$$\textit{rectangle-area}(3, 2, 6) \leftarrow \textit{number}(3), \textit{number(2)}$$

instead of $\textit{rectangle-area}(3, 2, 3 \times 2)$. On the other hand, there are functions that we do not want to evaluate in such a way. For example, if we have a pairing function $\textit{cons}(a, b)$ we want to leave it as it is. Intuitively, we will take the general approach that the functions are divided into two classes, interpreted built-in functions whose values are computed during the instantiation process and uninterpreted function symbols that are left as they are.

One of the main aims of this work is to create a formal framework for using cardinality constraint literals in conjunction with variables. The basic idea is that the semantics of a program P is defined with respect to a data model D. A program P contains two types of predicates: program and data. The idea is that the rules of the program define when the program predicates are true and the interpretations of the data predicates are given in the data model. In addition to data predicates, the data model also contains definitions of the universe of the problem instance and interpretations for all function symbols.

This approach is well-suited for developing *uniform encodings* [39] of problems. An uniform encoding of a problem is a single logic program that can be used to solve all instances of the problem by supplying the corresponding data models. For example, the general graph coloring problem has the following uniform encoding as a cardinality constraint program:

$$1 \leq \{\textit{has-color}(X, C) : \textit{color}(C)\} \leq 1 \leftarrow \textit{vtx}(X)$$
$$\leftarrow \textit{has-color}(X, C), \textit{has-color}(Y, C), \textit{edge}(X, Y), \textit{color}(C)$$

where the first rule demands that each vertex has exactly one color and the second rule requires that two adjacent vertices have different colors. The predicates *color*/1, *vtx*/1, and *edge*/2 are data predicates and *has-color*/2 is the only program predicate.

The construct $\textit{has-color}(X, C) : \textit{color}(C)$ is a *literal set* that denotes the set of atoms $\textit{has-color}(X, c)$ for which it holds that $\textit{color}(c)$ is true. For example, if there are three colors: red, blue, and green, then the literal set expands to the set:

$$\{\textit{has-color}(X, \textit{red}), \textit{has-color}(X, \textit{blue}), \textit{has-color}(X, \textit{green})\} \ .$$

Sometimes it is convenient to be able to define a part of the data model directly as rules in a program. For example, in Section 10.2 we will examine the Sokoban game that is basically a form of a planning puzzle where actions involve pushing boxes to different directions. There the data model involves the predicate $\textit{same-segment}(X_1, Y_1, X_2, Y_2, Dir)$ that is true if the squares (X_1, Y_1) and (X_2, Y_2) are along the same straight line in direction Dir with no structural obstruction between them. As there is a large number of possible lines in all but trivial cases, it would be impractical to define them all explicitly in the data model. On the other hand, *same-segment*/5 can be

easily defined using a set of rules of the form:

$$\begin{aligned} \textit{same-segment}(X, Y, X+1, Y, \textit{east}) &\leftarrow \textit{square}(X, Y), \textit{square}(X+1, Y) \\ \textit{same-segment}(X, Y, Z+1, Y, \textit{east}) &\leftarrow \textit{same-segment}(X, Y, Z, Y, \textit{east}) \\ &\quad \textit{square}(X, Y), \textit{square}(Z, Y), \\ &\quad \textit{square}(Z+1, Y) \ . \end{aligned}$$

We define a syntactic subclass of cardinality constraint programs, the ω-restricted programs that allows the programmer to define parts of the data model within the program. The predicate symbols that occur in the program are arranged into a hierarchy where the predicates on a higher level are defined in terms of predicates in the lower levels. A predicate is then a data predicate if it does not depend negatively on itself or on predicates that do. It is guaranteed that those predicates have the same extensions in all stable models of the program so interpreting them as data predicates does not cause any problems. Additionally, ω-restricted programs have the property that they stay decidable even when function symbols are allowed.

We examine the computational complexity of ω-restricted programs and it turns out it is equal to the case of normal logic programs so allowing cardinality constraints does not increase complexity. The program complexity for the ground case is **NP**-complete [51], **EXP**-complete for programs with variables and no non-constant function symbols, and 2-**EXP**-complete for programs with variables and function symbols. In the last case syntactic properties of ω-restricted programs ensure that the problem stays decidable.

In the course of this work we will also examine various ways to further generalize the notion of cardinality constraints. Two extensions that are perhaps the most intuitive ones are weight constraints [65] and classical negation [25]. A weight constraint works otherwise just as a cardinality constraint but each literal in it has a weight assigned to it and the constraint is satisfied if the sum of weights of satisfied literals is between the bounds. In programs with classical negation we allow an atom A to have two different kinds of negation, the default negation $\mathrm{not}(A)$ and the classical negation $\neg A$. The difference is that $\mathrm{not}(A)$ is true if cannot prove that A is true while $\neg A$ is true if we can prove that A is false.

We also consider a third extension, partial stable models [30]. Normal stable models are total in the sense that each atom of the program is either *true* or *false* in the model. Now we introduce a third truth value, *undefined*, to the semantics. Intuitively an atom is undefined in the model if both having it true and false lead into a contradiction, or if it depends on an undefined atom. Allowing partial models is particularly valuable when we are constructing encodings for our problems, since they can be used to identify errors in the rules. For example, the program:

$$\begin{aligned} 1 \leq \{a, b\} \leq 1 &\leftarrow c. \\ a &\leftarrow b \\ b &\leftarrow a \\ c &\leftarrow \end{aligned}$$

does not have a stable model but it does have a partial stable model where c is true and both a and b are undefined. This indicates that the contradiction

in the program is somehow related to the atoms a and b and it provides a starting point for debugging.[2]

The ω-restricted cardinality constraint programs have been implemented in the SMODELS system [48, 49, 50, 65, 77]. The system is divided into two parts, *smodels*, the actual inference engine, and *lparse*, a front end for instantiating the user programs. The input language of *smodels* does not include general cardinality constraints but they are instead translated into simpler rules that may be implemented efficiently.

1.1 Outline of the Work

We define the cardinality constraint programs formally in Section 3 and define stable model semantics for them in Section 4. We then introduce ω-restricted programs in Section 5. In Section 6 we generalize the notion of a cardinality constraint and present a few extensions to the semantics. We then show how the extensions can be translated back into standard ω-restricted programs in Section 7. We analyse the computational complexity in Section 8 and present some implementation details in Section 9. Finally, we show a few larger programming examples in Section 10.

1.2 Scientific Contributions

The main scientific contributions of this work are the formal stable model semantics for cardinality constraint programs with variables (Sections 3 and 4), and the definition of the class of ω-restricted cardinality constraint programs (Section 5).

Other contributions include a framework for generalizing cardinality constraint programs and a partial stable model semantics for cardinality constraint programs in Section 6, and analysis for computational complexity of ω-restricted programs in Section 8. The computational complexity results are partly based on previous work of the author [75].

[2] Here the first rule asserts that exactly one of a and b has to be true while the second two rules demand that both have to be true.

2 THE STABLE MODEL SEMANTICS OF NORMAL PROGRAMS

We start by giving the definition of the stable model semantics of propositional normal programs [24]. A *rule* is an expression of the form:

$$h \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m$$

where h, a_i, and b_i are all propositional atoms. A rule is *fact* if $n = 0$ and $m = 0$, and it is a Horn rule if $m = 0$. A *logic program* P is a set of rules.

As Horn constraint rules are monotonic, a program consisting only of such rules has a unique minimal model [79]. We can find this model by using the operator T_P. Let P be a Horn program and S be a set of atoms occurring in P. Then $T_P(S)$ is defined as follows:

$$T_P(S) = \{h \mid h \leftarrow a_1, \ldots, a_n \in P \text{ and } \{a_1, \ldots, a_n\} \subseteq S\} \ .$$

The minimal model is the least fixed point of T_P [79].

Example 2.1 *Let P be the program:*

$$\begin{array}{ll} a \leftarrow b, c & d \leftarrow \\ b \leftarrow d & e \leftarrow f \\ c \leftarrow \ . & \end{array}$$

Now, the computation of the minimal model proceeds as follows:

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

Thus, the minimal model of P is $M = \{a, b, c, d\}$.

Given a set of atoms S, the *reduct* P^S of a program P with respect to S is obtained by removing from P:

1. Every rule r such that the body of r contains a negative literal $\text{not}(a)$ where $a \in S$; and

2. All negative literals from bodies of remaining rules.

This process may be viewed as replacing a negative literal with the truth value that it has in S so that a true one is replaced by T and a false one by F.

Now, a set of atoms S is a *stable model* of P if and only if S is the minimal model the reduct P^S.

Example 2.2 *Let P be the program:*

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

Now, P has two stable models: $M_1 = \{a, c\}$ and $M_2 = \{b\}$. We see that the reduct P^{M_1} is the set of rules:

$$a \leftarrow$$
$$c \leftarrow a$$

and the least fixed point of $T_{P^{M_1}}$ is, indeed, $\{a, c\}$. The set $M_3 = \{a, b, c\}$ is a model of P in the classical sense, but it is not stable since $P^{M_3} = \emptyset$ whose minimal model is also empty.

The properties of the reduct ensure that S is a model of P^S if and only if it is also a model of the original program P. When we define the stable semantics for cardinality constraint programs we see that this condition does not hold there, and we have to add an extra condition and explicitly state that S has to be also a model of P.

3 CARDINALITY CONSTRAINT PROGRAMS

The basic component of a logic program is an *atom* of the form:

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

where p is a n-ary predicate symbol ($n \geq 0$) and t_1, ..., t_n are terms. A *term* is either a variable v, a constant c, or an m-ary function symbol $f(t_1, \ldots, t_m)$ where t_1, ..., t_m are terms. A 0-ary function symbol is a *constant*. The set of all available function symbols is denoted by $\mathcal{F}$. We denote the predicate symbol of an atom a by $pred(a)$. A *basic literal* is either an atom a or its negation $\text{not}(a)$. The set of all predicate symbols $\mathcal{P}$ is divided into two classes, *program* ($\mathcal{P}_P$) and *data* ($\mathcal{P}_D$) predicates. Program predicates are those whose values are defined by the program while extensions of data predicates are fixed and come from some external data model. We demand that $\mathcal{P}_P$ should be finite. We also define a special atom, $\top$ that is always true. For simplicity, we allow $\top$ to occur as both a program and a data predicate.

A *literal set* S is of the form:

$$\rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m \rangle \tag{3}$$

where X_1, ..., X_n are the *local variables*, the basic literal l is the *main literal*, and the basic atoms $a_1, \ldots, a_m$ are *conditions*. The main literal has to be a program predicate and only data predicates may occur among conditions. Intuitively, the notation $a_1, \ldots, a_n$ denotes the conjunction of the atoms a_i. In theory, we could replace the sequence by a single atom a where $pred(a)$ is a new data predicate whose extension is defined accordingly. However, we do not do that because there are cases where the more verbose notation is clearer.

We can suppose without loss of generality that none of the local variables occur elsewhere in a program since we can always systematically rename them.[3] We denote the local variables, main literal, and the set of conditions of S respectively by $vars_{\mathfrak{L}}(S)$, $lit(S)$, and $conds(S)$. We say that S is positive if l is an atom a, otherwise S is negative.

The main idea of S of the form (3) is that it denotes the set of basic literals l' that is obtained by substituting a ground term for each local variable $X \in vars_{\mathfrak{L}}(S)$ in such a way that each literal a_i is satisfied with the substitution . We call this set the *expansion* of S and denote it with $E(S)$. We present the formal definition in the next section.

Example 3.1 *Consider the literal set $S = \rho X.\langle p(X) : q(X) \rangle$. Then, S denotes the set of literals $E(S) = \{p(a) \mid q(a)$ is true in the data model$\}$.*

A *constraint literal* C is of the form:

$$L \leq \{S_1, \ldots, S_n\} \leq U$$

where L and U are integral upper and lower *bounds* and S_1, ..., S_n are literal sets. One or both bounds may be left out. In those cases we take 0 and ∞ to

[3] However, in some example programs we use the same local variable in many literal sets when the variables have a similar function.

be the bounds. We use the notation $bound_L(C) = L$, $bound_U(C) = U$, and $\mathcal{L}(C) = \{S_1, \ldots, S_n\}$. However, we will often abuse the notation by using $S \in C$ in place of $S \in \mathcal{L}(C)$ in places where there is no risk of confusion.

The intuition is that a constraint literal is true whenever the number of basic literals true in the union of the expansions of literal sets S_i is between L and U, inclusive. For example, $1 \leq \{\rho X.\langle \mathit{select}(X) : \mathit{vtx}(X)\rangle\} \leq 1$ is true when exactly one atom $\mathit{select}(v)$ is true for which $\mathit{vtx}(v)$ is also true.

The set of basic literals belonging to an expanded constraint literal is just that, a set. It would be possible to extend the semantics to allow it to be a multiset, but experience has shown that the set semantics is more intuitive when encoding problems into cardinality constraint programs.

We use shorthand notations for several forms of constraint literals that behave analogously to basic literals in normal logic programs. We use an atom a to denote a constraint literal that is either of the form $1 \leq \{a : \top\}$ when $pred(a)$ is a program predicate, or $1 \leq \{\top : a\}$ if $pred(a)$ is a data predicate. Similarly, $\mathrm{not}(a)$ denotes constraint literals $1 \leq \{\mathrm{not}\ a : \top\}$ and $1 \leq \{\top : \mathrm{not}\ a\}$.

A *rule* r is of the form:

$$C_0 \leftarrow C_1, \ldots, C_n$$

where all C_i are constraint literals. Here C_0 is the *head* and C_1, ..., C_n form the *body*. We say that a rule is *simple* if $bound_L(C_0) = 1$, $\mathcal{L}(C_0) = \{\langle h : \top\rangle\}$ for some atom h, and $bound_U(C_i) = \infty$ for each $i \geq 0$. A simple rule is a *Horn constraint rule* if all main literals that occur in it are positive. A *logic program* P is a possibly infinite set of rules. The sets of literal sets occurring in a rule r or a program P are denoted by $\mathcal{L}(r)$ and $\mathcal{L}(P)$, respectively.

Intuitively a rule asserts that if all constraint literals in the rule body are satisfied, then the head must be true also. We will also allow rules with empty heads that prune out unwanted model candidates; if the body of the rule is true, then we get a contradiction. We can interpret such a rule as a shortcut for a rule with an unsatisfiable head of the form: $2 \leq \{\langle f : \top\rangle\}$ where f does not occur anywhere else in the program[4].

The set $vars(t)$ of variables that *occur in* a term is defined as follows:

$$vars(t) = \begin{cases} \emptyset & \text{, if } t \text{ is a constant;} \\ \{t\} & \text{, if } t \text{ is a variable; and} \\ \bigcup_{i=1}^{m} vars(t_i) & \text{, if } t \text{ is a function } f(t_1, \ldots, t_m)\ . \end{cases}$$

A variable occurs in a basic literal l if it occurs in at least one of its arguments:

$$vars(l(t_1, \ldots, t_m)) = \bigcup_{i=1}^{m} vars(t_i)\ .$$

A variable occurs in a literal set S if it occurs either in the main literal or

[4]When we deal with a general cardinality constraint program, having f occur in also other places does not cause problems. However, it may problems when ω-restricted programs (as defined in Section 5) are used, so we forbid it also in the general case.

in some condition and it is not a local variable:

$$vars(\rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m \rangle) = \\ (vars(l) \cup \bigcup_{i=1}^{m} vars(a_i)) \setminus \{X_1, \ldots, X_n\} \tag{4}$$

A literal set is *ground* if $n = 0$ and $vars(S) = \emptyset$.

In a similar manner the set of variables that occur in a constraint literal C, a rule r, or a program P is defined to be the union of the sets of variables that occur in their components:

$$\begin{aligned} vars(C) &= \bigcup_{S \in \mathcal{L}(C)} vars(S) \\ vars(r) &= \bigcup_{i=0}^{n} vars(C_i) \\ vars(P) &= \bigcup_{r \in P} vars(r) \quad . \end{aligned}$$

A constraint literal is ground if all literal sets in it are, and a rule is ground if all constraint literals in it are.

4 STABLE MODEL SEMANTICS OF CARDINALITY CONSTRAINT PROGRAMS

In this section we define the stable model semantics for cardinality constraint programs. The basic semantics is defined directly for the ground programs, and rules with variables are seen to denote the sets of their ground instances. The instantiation is done relative to a data model. In some cases we might use the Herbrand interpretation of the program as our model, but there are cases where we want the data model to be more complex.

The definitions in this section are more complex than the corresponding ones for normal logic programs. There are two main reasons for this:

1. we want to allow the use of some evaluated functions, for example arithmetic operators, during the instantiation process; and

2. we have two kinds of variables, local and global.

We use the notation $\langle P, D \rangle$ to denote the combination of a program P and a data model D. We do not restrict the ways how the data model may be defined; in the simplest case it might be a set facts, or it might come from a relational database.

The basic motivation for having a data model is that it provides a set of built-in functions and predicates. For example, the data model may define the basic arithmetic functions on integers ($+$, $-$, ...) as well as some predicates ($<$, $\leq$, $>$, ...) that can be used to make programs more compact.

As usual in logic programming we give the basic semantics for ground programs and rules with variables are seen as short-hand notations for denoting the set of their ground instances. However, this interpretation is not as straightforward as in most other cases since literal sets and their local variables have to be handled carefully. In practice we have to do the instantiation in two stages where we instantiate all global variables and then expand all literal sets to remove local variables.

We define the semantics using a Gelfond-Lifschitz-style [24] reduction that transforms a cardinality constraint program P into a simpler one that has a unique minimal model M'. The reduction is done with respect to some set of atoms M. If it happens so that M satisfies P and coincides with M', then M is a stable model of $\langle P, D \rangle$. Our definition of the reduct extends the definition given in [65] by allowing the use of literal sets.

The definition of the stable model semantics is divided into three parts:

1. *instantiation* where global variables are removed from the rules;

2. *expansion* where each literal set with local variables in P is replaced by a set of ground basic literals; and

3. *reduction* where each rule of P is replaced by a possibly empty set of Horn constraint rules.

Next we will give the definitions for data models and the three parts of stable model semantics. We will use the Hamiltonian cycle problem as a running example:

Example 4.1 *In the Hamiltonian cycle problem we are given a directed graph $\langle V, E\rangle$ and an initial vertex $i \in V$ and the object is to find a cycle that visits each vertex exactly once. This problem can be given the following uniform encoding:*

$$\begin{aligned} 1 \leq \{\rho Y.\langle hc(X,Y) : edge(X,Y)\rangle\} \leq 1 &\leftarrow vtx(X) \\ 1 \leq \{\rho Y.\langle hc(Y,X) : edge(Y,X)\rangle\} \leq 1 &\leftarrow vtx(X) \\ r(Y) \leftarrow 1 \leq \{r(X), initial(X)\}, hc(X,Y), edge(X,Y) & \\ \leftarrow 1 \leq \{\rho X.\langle \text{not } r(X) : vtx(X)\rangle\} & \end{aligned} \tag{5}$$

The first two rules ensure that there is exactly one incoming and one outgoing edge for each vertex. The last two rules ensure that each vertex is visited by the path. Here, the predicates edge/2, vtx/1, and initial/1 are data predicates defined in the data model, and the predicates hc/2 and r/1 are program predicates.

4.1 Data Models

The semantics of a program P is given relative to a data model $D = \langle U, \Sigma, I\rangle$ where U is the *universe*, $\Sigma = \langle \mathcal{P}_D, \mathcal{F}, \mathcal{N}\rangle$ is the *vocabulary* consisting of a set of *data predicate symbols* $\mathcal{P}_D$, a set of *function symbols* $\mathcal{F}$, a set of 0-ary *names* $\mathcal{N} \subseteq \mathcal{F}$, and an interpretation I that assigns:

1. a relation $I(p) \subseteq U^k$ for each k-ary predicate symbol $p \in \mathcal{P}_D$; and
2. a computable mapping $I(f) : U^k \to U$ for each k-ary function symbol $f \in \mathcal{F}$.

An interpretation I assigns a constant function $I(c) = \varepsilon \mapsto \mathbf{u}$ for each 0-ary constant symbol $c \in \mathcal{F}$. In these cases we will identify this function with its value and simply denote that $I(c) = \mathbf{u}$.

The set $\mathcal{N}$ of names has to contain a unique 0-ary name for each element $\mathbf{u} \in U$. That is, the interpretation I has to map each name $n \in \mathcal{N}$ to a different element of U and for each element $\mathbf{u} \in U$ there has to be an element $n \in \mathcal{N}$ such that $I(n) = \mathbf{u}$. We will denote the inverse function of I restricted to $\mathcal{N}$ by N so that $N(I(n)) = n$ for each $n \in \mathcal{N}$. Note that I restricted to the set of all 0-ary constants does not have to be injective and it may map more than one constant to the same element of U.

The data model provides the universe and the interpretations for all data predicates and function symbols that may occur in a program. The reason for making a distinction between the elements of U and their names in $\mathcal{N}$ is that we can then separate the syntax of a program from the semantics given by U and I when handling quantification of variables in the rules.

We do not impose restrictions on the nature of U; it may be a finite set of elements, or perhaps an infinite set of terms. However, for the purpose of defining the semantics we disregard any possible internal structure of elements of U. In this section we will use the convention that the elements of U are written using bold-face (e.g. $\mathbf{0}$, $\mathbf{f}(\mathbf{a})$) while all syntactic elements are written using italic font (e.g. 0, $f(a)$). Unless specified otherwise, the name of an element $\mathbf{x}$ is defined as $N(\mathbf{x}) = \underline{x}$. For example, $N(\mathbf{f}(\mathbf{a})) = \underline{f(a)}$.

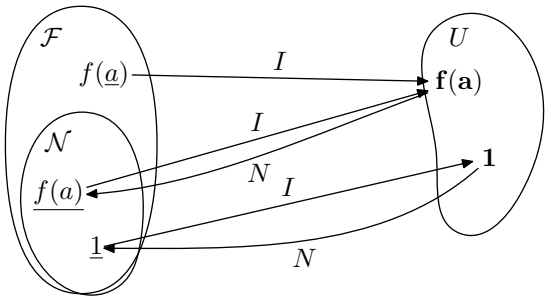


Figure 1: The relationship of function symbols $\mathcal{F}$, names $\mathcal{N}$, and the universe U.

Note that the naming function is arbitrary and we do not in general assume that the name of $\mathbf{f}(\mathbf{a})$ has to have any connection to the symbols f and a.

Here we want to emphasize that the different notations for $f(a)$ are used to make it more clear that the string of four characters '$f(a)$' can occur in three different roles:

1. it may denote the application of a function symbol f to the ground term a;

2. it may denote the corresponding element of the universe; or

3. it may denote the name of the element.

Later on we will blur the distinction between names, terms, and elements and use $f(a)$ in all three roles.

The interpretation of a k-ary function symbol $f \in \mathcal{F}$ is total so it has to be defined for each possible combination of arguments. The interpretation of a compound term is defined in the standard way by using bottom-up evaluation for interpreting its arguments.

Definition 4.1 *The* interpretation $I(f(t_1, \ldots, t_k))$ *of a term* $(k \geq 0)$ *under a data model* $\langle U, \langle \mathcal{P}_D, \mathcal{F}, \mathcal{N} \rangle, I \rangle$ *is the element:*

$$I(f(t_1, \ldots, t_k)) = I(f)(I(t_1), \ldots, I(t_k)) \ . \tag{6}$$

An interpretation I imposes a satisfaction relation for ground data atoms.

Definition 4.2 *Given a data model* $D = \langle U, \langle \mathcal{P}_D, \mathcal{F}, \mathcal{N} \rangle, I \rangle$*, a ground atom* $p(t_1, \ldots, t_k)$*,* $p \in \mathcal{P}_D$*, is* satisfied *by* D *(denoted* $D \models p(t_1, \ldots, t_k)$*) if and only if* $\langle I(t_1), \ldots, I(t_k) \rangle \in I(p)$*, and* $D \models \mathrm{not}(p(t_1, \ldots, t_k))$ *if and only if* $D \not\models p(t_1, \ldots, t_k)$*.*

Next, we give a few examples on how a data model may be defined and how interpretations work. In the first example we define a simple data model.

Example 4.2 *Let $D = \langle U, \langle \mathcal{P}_D, \mathcal{F}, \mathcal{N} \rangle, I \rangle$ be a data model where:*

$$\begin{aligned} U &= \{\mathbf{0}, \mathbf{1}, \mathbf{2}\} \\ \mathcal{P}_D &= \{\mathit{greater}/2\} \\ \mathcal{F} &= \{+/2, \times/2, 0, 1, 2\} \cup \mathcal{N} \\ \mathcal{N} &= \{\underline{0}, \underline{1}, \underline{2}\} \ . \end{aligned}$$

We want to define the interpretation I so that the function $+$ corresponds to addition modulo 3, and $\times$ to multiplication modulo 3. We start by defining the interpretation for the six constants (three names and three other constants) of the model:

$$\begin{aligned} I(0) &= I(\underline{0}) = \mathbf{0} \\ I(1) &= I(\underline{1}) = \mathbf{1} \\ I(2) &= I(\underline{2}) = \mathbf{2} \ . \end{aligned}$$

Strictly speaking it is not necessary to separate a constant x from the name $\underline{x}$ since the interpretations of constants 0, 1, and 2 already satisfy the naming condition. However, they are included in the data model to illustrate the situation that more than one constant may be mapped to the same element of U.

The interpretation of a k-ary function symbol $f \in \mathcal{F}$ is some function $U^k \to U$. Thus, we set $I(+) = h$ where h is defined as follows:

$$h(\mathbf{x}, \mathbf{y}) = \mathbf{x} + \mathbf{y} \mod 3 \ .$$

Similarly, $I(\times) = g$ where $g(\mathbf{x}, \mathbf{y}) = \mathbf{x} \times \mathbf{y} \mod 3$.

The interpretation of the binary predicate greater/2 is defined as a set of pairs of elements:

$$I(\mathit{greater}) = \{\langle \mathbf{x}, \mathbf{y} \rangle \mid \mathbf{x}, \mathbf{y} \in U \text{ and } \mathbf{x} > \mathbf{y}\} \ .$$

In the second example we evaluate some ground terms and data atoms using the simple data model.

Example 4.3 *Let D be the data model defined in Example 4.2. We want to find the truth values for two ground atoms:*

$$\begin{aligned} A &= \mathit{greater}(1, 2) \\ B &= \mathit{greater}((2 \times 2) + 1, 1 + 2) \ . \end{aligned}$$

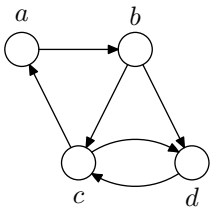
To see whether a data atom is satisfied in a data model or not, we have to start by finding the interpretations of the arguments. In the first case we have:

$$\begin{aligned} I(1) &= \mathbf{1} \\ I(2) &= \mathbf{2} \ . \end{aligned}$$

As $\mathbf{1} < \mathbf{2}$, $\langle \mathbf{1}, \mathbf{2} \rangle \notin I(\mathit{greater})$ and $D \not\models A$.

Consider now the first argument of B. When we use a prefix notation the term becomes $+(\times(2, 2), 1)$. By Definition 4.1,

$$I(+(\times(2, 2), 1)) = I(+)(I(\times(2, 2)), I(1)) \ .$$



$$
\begin{aligned}
D &= \langle U, \Sigma, I \rangle \\
U &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\} \\
\Sigma &= \langle \{\mathit{vtx}/1, \mathit{edge}/2, \mathit{initial}/1\}, \mathcal{N}, \mathcal{N} \rangle \\
\mathcal{N} &= \{\underline{a}, \underline{b}, \underline{c}, \underline{d}, \underline{e}\} \\
I(\mathit{vtx}) &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\} \\
I(\mathit{edge}) &= \{\langle \mathbf{a}, \mathbf{b} \rangle, \langle \mathbf{b}, \mathbf{c} \rangle, \langle \mathbf{b}, \mathbf{d} \rangle, \langle \mathbf{c}, \mathbf{a} \rangle, \\
&\qquad \langle \mathbf{c}, \mathbf{d} \rangle, \langle \mathbf{d}, \mathbf{c} \rangle\} \\
I(\mathit{initial}) &= \{\mathbf{a}\} \\
I(\underline{a}) &= \mathbf{a}; I(\underline{b}) = \mathbf{b}; I(\underline{c}) = \mathbf{c}; \\
I(\underline{d}) &= \mathbf{d}; I(\underline{e}) = \mathbf{e}
\end{aligned}
$$

Figure 2: A sample data model for the Hamiltonian cycle example.

. *Continuing the evaluation of* I *we get:*

$$
\begin{aligned}
I(+(\times(2,2),1)) &= I(+)(I(\times(2,2)), I(1)) \\
&= h(I(\times)(I(2), I(2)), \mathbf{1}) \\
&= h(g(\mathbf{2}, \mathbf{2}), \mathbf{1}) \\
&= h(\mathbf{1}, \mathbf{1}) \\
&= \mathbf{2} \ .
\end{aligned}
$$

For the second argument the evaluation proceeds as:

$$
\begin{aligned}
I(+(1,2)) &= I(+)(I(1), I(2)) \\
&= h(\mathbf{1}, \mathbf{2}) \\
&= \mathbf{0} \ .
\end{aligned}
$$

As $\mathbf{2} > \mathbf{0}$, $\langle \mathbf{2}, \mathbf{0} \rangle \in I(\mathit{greater})$ *so* $D \models B$.

Before we delve deeper into the realm of function symbols and their interpretations, we present another simple data model that will be used later in this section in the examples of rule instantiations.

Example 4.4 *A sample data model for the Hamiltonian cycle problem that was introduced in Example 4.1 is presented in Figure 2.*

Herbrand Interpretations

Example 4.2 showed us how we can use functions that compute some concrete value in a data model. However, the standard practice in logic programming is to use *Herbrand interpretation* for function symbols. There the universe consists of all terms that can be formed by using the function symbols and constants that occur in the program. A Herbrand universe is always infinite if there exists at least one function symbol with arity greater than zero. Next, we show how we can construct a data model that corresponds to the Herbrand interpretation of the program.

Suppose that we have some program P and we want to use its Herbrand interpretation. The actual literals and rules of P are not relevant when we construct the interpretation, only the constants and other function symbols

that occur in it. In practice, we have to define the sets U and $\mathcal{N}$ as well as the interpretation I.

First, we define the sets C and F of function symbols that occur in P:

$$\begin{aligned} C &= \{c \mid c \text{ is a 0-ary function symbol (constant) occuring in } P\} \\ F &= \{f \mid f \text{ is a } k\text{-ary } (k > 0) \text{ function symbol occurring in } P\} \ . \end{aligned} \tag{7}$$

Then, we can define the set of names $\mathcal{N}$. The intuition is that we add each Herbrand term that can be constructed using C and F as a name in $\mathcal{N}$. The set $\mathcal{N}$ is the smallest set that fulfills the following two conditions:

1. For all $c \in C$, $\underline{c} \in \mathcal{N}$.
2. For all $t_1, \ldots, t_k \in \mathcal{N}$ and $f \in F$, $\underline{f(t_1, \ldots, t_k)} \in \mathcal{N}$.

As the definition leads to names with many underlines, we sometimes simplify the notation and leave out the inner underlines. For example,

$$\underline{f(\underline{g(\underline{a})}, \underline{a})} = \underline{f(g(a), a)} \ .$$

Next, we define the universe U so that it contains exactly one element for each name in $\mathcal{N}$. The simplest way to do it to say that:

$$U = \{\mathbf{x} \mid x \in \mathcal{N}\} \ . \tag{8}$$

Note that with this definition we end up with a situation where the universe contains elements whose syntactic representations are underlined. We do this because it makes it easier to define the interpretations of function symbols.

Finally, we have to construct the interpretation I. We see that I should map terms occurring in P to the corresponding Herbrand terms in U and to do that in a consistent way. Thus, we have to ensure that, for example, $I(f(a)) = I(f(\underline{a})) = I(\underline{f(\underline{a})})$. This can be achieved by defining $I(f)$ to be the function f':

$$f'(\mathbf{x}) = \underline{\text{`}\mathbf{f}\text{'} \circ \text{`(' } \circ \mathbf{x} \circ \text{`)'}} \tag{9}$$

where $\circ$ denotes the string concatenation. For all names $\underline{n} \in \mathcal{N}$, we set $I(\underline{n}) = \underline{\mathbf{n}}$, and for all other constants $c \in \mathcal{F}$, we set $I(c) = \underline{\mathbf{c}}$.

We will use the notation $D_{P,H}$ later on to denote the data model that corresponds to the Herbrand interpretation of P.

Example 4.5 *Consider the case where we have two constants, a and b, and one unary function symbol, f, occurring in a program and we want to use the Herbrand interpretation for it. Then, the set of names $\mathcal{N}$ is the infinite set:*

$$\mathcal{N} = \{\underline{a}, \underline{b}, \underline{f(\underline{a})}, \underline{f(\underline{b})}, \underline{f(\underline{f(\underline{a})})}, \ldots\} \ .$$

The universe is then the set U:

$$U = \{\underline{\mathbf{a}}, \underline{\mathbf{b}}, \underline{\mathbf{f}(\underline{\mathbf{a}})}, \underline{\mathbf{f}(\underline{\mathbf{b}})}, \underline{\mathbf{f}(\underline{\mathbf{f}(\underline{\mathbf{a}})})}, \ldots\} \ .$$

The interpretation I is defined as above. Suppose that we have to interpret the term $f(f(\underline{a}))$. Then,

$$\begin{aligned} I(f(f(\underline{a}))) &= I(f)(I(f(\underline{a}))) \\ &= f'(I(f)(I(\underline{a}))) \\ &= f'(f'(\underline{\mathbf{a}})) \\ &= f'(\underline{\mathbf{f}(\underline{\mathbf{a}})}) \\ &= \underline{\mathbf{f}(\underline{\mathbf{f}(\underline{\mathbf{a}})})} \ . \end{aligned}$$

Thus, we can use "uninterpreted" function symbols by defining their interpretation suitably.

Combining Herbrand Interpretations and Evaluated Functions
We can also define the data model so that some of the function symbols have the Herbrand interpretation while the others are evaluated as in Example 4.2. However, as interpretations are total, we have to define the value for nonsensical applications of function, for example, $f(b) + 5$ has to evaluate to some element of the universe. There are two basic approaches for this:

1. we can add a new element $\mathbf{e}$ to the universe to denote the error condition and define $I(x + y)$ to be $\mathbf{e}$ whenever $I(x)$ and $I(y)$ are not both natural numbers; or

2. we can use Herbrand interpretation for them.

In a practical implementation of the semantics we might want to take the third approach and signal a type error to the programmer.

Example 4.6 *Consider the following data model D:*

$$\begin{aligned} D &= \langle U, \Sigma, I \rangle \\ U &= \{\underline{\mathbf{e}}, \underline{\mathbf{0}}, \underline{\mathbf{1}}, \underline{\mathbf{f}(\mathbf{e})}, \underline{\mathbf{f}(\mathbf{0})}, \underline{\mathbf{f}(\mathbf{1})}, \underline{\mathbf{f}(\mathbf{f}(\mathbf{e}))}, \ldots\} \\ \Sigma &= \{\emptyset, \mathcal{F}, \mathcal{N}\} \\ \mathcal{F} &= \{+/2, f/1, 0, 1, e\} \cup \mathcal{N} \\ \mathcal{N} &= \{\underline{e}, \underline{0}, \underline{1}, \underline{f(e)}, \underline{f(0)}, \underline{f(1)}, \underline{f(f(e))}, \ldots\} \end{aligned}$$

where we want $f/1$ to have a Herbrand interpretation and $+/2$ to define addition modulo 2. The definition of the interpretation of f can be defined as in Example 4.5, that is $I(f) = f'$ where $f'(\mathbf{x}) = \underline{\text{`}\mathbf{f}\text{'} \circ \text{`(`} \circ \mathbf{x} \circ \text{`)'}}$.

For $+$ we say that $I(+) = h$ where:

$$h(\mathbf{x}, \mathbf{y}) = \begin{cases} \mathbf{x} + \mathbf{y} \mod 2, & \text{if } \{\mathbf{x}, \mathbf{y}\} \subseteq \{\underline{\mathbf{0}}, \underline{\mathbf{1}}\} \\ \mathbf{e}, & \text{otherwise} \ . \end{cases}$$

The intuition is that $\underline{\mathbf{e}}$ acts as an error value that denotes that the value of the function is undefined for x and y. Finally, $I(e) = \underline{\mathbf{e}}$, $I(0) = \underline{\mathbf{0}}$, $I(1) = \underline{\mathbf{1}}$, and for each $\underline{x} \in \mathcal{N}$, $I(\underline{x}) = \underline{\mathbf{x}}$.

Consider now a compound term $f(0+1)$ and its interpretation. By Definition 4.1:

$$\begin{aligned} I(f(0+1)) &= I(f)(I(0+1)) \\ &= f'(I(0+1)) \\ &= f'(I(+)(I(0), I(1))) \\ &= f'(h(\underline{\mathbf{0}}, \underline{\mathbf{1}})) \\ &= f'(\underline{\mathbf{1}}) \\ &= \underline{\mathbf{f(1)}} \ . \end{aligned}$$

4.2 Removal of Global Variables

We start the formal definitions by the concept of a substitution where variables are replaced by ground terms.

Definition 4.3 *Let P be a cardinality constraint program, ρ be a set of variables and $D = \langle U, \Sigma, I \rangle$ a data model. Then, a* substitution *$\sigma_{\rho,D}$ is a function*

$$\sigma_{\rho,D} : \rho \to \mathcal{N} \tag{10}$$

that associates an element $\sigma_{\rho,D}(v) \in \mathcal{N}$ for each variable $v \in \rho$. We denote the set of all substitutions on ρ and D by $subs(\rho, D)$.

For clarity, we will often leave out the subscripts of a substitution where there is no danger of confusion. We will use the notation $X/\underline{a}$ to denote that a substitution maps the variable X to the name $\underline{a}$.

Example 4.7 *Let $\sigma_{\rho,D} = \{x \mapsto \underline{a}, y \mapsto \underline{b}, z \mapsto \underline{f(c,a)}\}$ be a substitution. Then, an alternate representation would be $\{x/\underline{a}, y/\underline{b}, z/\underline{f(c,a)}\}$.*

Definition 4.4 *Given a substitution $\sigma \in subs(\rho, D)$, a* (partial) instantiation *of a term t with respect to σ (denoted $t\sigma$) is defined recursively as follows:*

$$t\sigma = \begin{cases} \sigma(t), & \text{if } t \text{ is a variable and } t \in \rho \\ t, & \text{if } t \text{ is a variable and } t \notin \rho \\ N(I(f)(I(t_1\sigma), \ldots, I(t_n\sigma))), & \text{if } t = f(t_1, \ldots, t_n) \text{ and} \\ & vars(f(t_1\sigma, \ldots, t_n\sigma)) = \emptyset \\ f(t_1\sigma, \ldots, t_n\sigma), & \text{if } t = f(t_1, \ldots, t_n) \text{ and} \\ & vars(f(t_1\sigma, \ldots, t_n\sigma)) \neq \emptyset \end{cases} \tag{11}$$

An instantiation is total *or* ground *if $vars(t) \subseteq \rho$.*

Note that in the above a constant is a 0-ary function symbol.

Definition 4.5 *Given a substitution $\sigma \in subs(\rho, D)$, a* (partial) instantiation *of an atom $p(t_1, \ldots, t_k)$ (denoted $p(t_1, \ldots, t_k)\sigma$) is the atom $p(t_1\sigma, \ldots, t_k\sigma)$.*

The partial instantiations of basic literals, literal sets, constraint literals, are done in the same way, by instantiating all terms in them.

Example 4.8 *Consider an atom $A = p(f(X+5), Y)$ where $+$ is evaluated and f has the Herbrand interpretation, and a substitution $\sigma = \{X/\underline{3}\}$. When we apply σ to A, we do it in a bottom-up fashion by applying it first to the innermost expressions that occur in A. There, we see that:*

$$\begin{aligned} X\sigma &= \underline{3} \\ 5\sigma &= N(I(5)) = N(\mathbf{5}) = \underline{5} \\ Y\sigma &= Y \ . \end{aligned}$$

Next, we examine the term $X + 5$:

$$\begin{aligned} (X+5)\sigma &= N(I(+)(I(X\sigma), I(5\sigma))) \\ &= N(I(+)(I(\underline{3}), I(\underline{5})) \\ &= N(I(+)(\mathbf{3}, \mathbf{5})) \\ &= N(\mathbf{8}) \\ &= \underline{8} \ . \end{aligned}$$

Moving one step up, we find that:

$$\begin{aligned} f(X+5)\sigma &= N(I(f)(I(X+5)\sigma)) \\ &= N(I(f)(\mathbf{8})) \\ &= N(\mathbf{f}(\mathbf{8})) \\ &= \underline{f(8)} \ . \end{aligned}$$

Thus,

$$p(f(X+5, Y)\sigma = p(\underline{f(8)}, Y) \ .$$

Example 4.9 *Let r be the rule*

$$1 \leq \{\langle a(X, Y+2) : b(X, Y)\rangle\} \leftarrow 1 \leq \{\langle c(X) : \top\rangle, \langle d(f(Y)) : e(Y)\rangle\}$$

where $+/2$ is evaluated, $f/1$ has Herbrand interpretation, and $\sigma = \{Y/\underline{1}\}$ a substitution. Then, the partial instantiation $r\sigma$ is the rule:

$$1 \leq \{\langle a(X, \underline{3}) : b(X, \underline{1})\rangle\} \leftarrow 1 \leq \{\langle c(X) : \top\rangle, \langle d(\underline{f(1)}) : e(\underline{1})\rangle\} \ .$$

A partial instantiation of a program P with respect to D is obtained by replacing each rule r by the set of all instances that can be generated by substituting elements from the set $\mathcal{N}$ of names for each global variable in r.

Definition 4.6 *The* partial instantiation *of a cardinality constraint program P with respect to a data model D is the set*

$$\mathit{inst}(P, D) = \{r\sigma \mid r \in P \text{ and } \sigma \in \mathit{subs}(\mathit{vars}(P), D)\} \ . \tag{12}$$

Note that this definition leaves all local variables untouched, since by (4) they do not belong to $\mathit{vars}(r)$.

Example 4.10 *The first rule of* (5) *in Example 4.1 is instantiated to:*

$$\begin{aligned}
&1 \leq \{\rho Y.\langle hc(\underline{a}, Y) : edge(\underline{a}, Y)\rangle\} \leq 1 \leftarrow vtx(\underline{a}) \\
&1 \leq \{\rho Y.\langle hc(\underline{b}, Y) : edge(\underline{b}, Y)\rangle\} \leq 1 \leftarrow vtx(\underline{b}) \\
&1 \leq \{\rho Y.\langle hc(\underline{c}, Y) : edge(\underline{c}, Y)\rangle\} \leq 1 \leftarrow vtx(\underline{c}) \\
&1 \leq \{\rho Y.\langle hc(\underline{d}, Y) : edge(\underline{d}, Y)\rangle\} \leq 1 \leftarrow vtx(\underline{d}) \\
&1 \leq \{\rho Y.\langle hc(\underline{e}, Y) : edge(\underline{e}, Y)\rangle\} \leq 1 \leftarrow vtx(\underline{e}) \ .
\end{aligned} \tag{13}$$

The five substitutions that give rise to these rules are: $\{X/\underline{a}\}$, $\{X/\underline{b}\}$, $\{X/\underline{c}\}$, $\{X/\underline{d}\}$ *and* $\{X/\underline{e}\}$.

4.3 Expansion

The expansion removes all local variables from a cardinality constraint program. Each literal set is replaced by a set of basic literals. The set is constructed by applying all substitutions that satisfy all conditions to the main literal.

Definition 4.7 *The* expansion *of a literal set* $S = \rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m\rangle$ *with respect to a data model* D *is the set*

$$E(S, D) = \{l\sigma \mid \sigma \in subs(vars_{\mathfrak{L}}(S), D) \wedge \forall i \in [1, m] : D \models a_i\sigma\} \ . \tag{14}$$

A constraint literal is expanded by expanding all literal sets in it.

Definition 4.8 *The* expansion *of a constraint literal*

$$C = L \leq \{S_1, \ldots, S_n\} \leq U$$

with respect to a data model D *is defined as follows:*

$$E(C, D) = L \leq E(S_1, D) \cup \cdots \cup E(S_n, D) \leq U \ . \tag{15}$$

We say that $E(C, D)$ *is an* expanded constraint literal.

The difference between unexpanded and expanded constraint literals is that an unexpanded one contains literal sets while an expanded one contains only basic literals. Note that after expansion there are only program predicates left in the program. All simple data constraint literals $1 \leq \{\langle \top : a\rangle\}$ are replaced by $1 \leq \{\top\}$ if a is true, or by an unsatisfiable literal $1 \leq \{\}$ if a is false.

Example 4.11 *Consider the constraint literal* C*:*

$$C = 1 \leq \{\rho X.\langle a(X) : d_1(X)\rangle, \rho Y.\langle a(Y) : d_2(Y)\rangle\} \leq 2 \ .$$

Suppose that the interpretations of d_1 *and* d_2 *in a data model* D *are:*

$$\begin{aligned}
I(d_1) &= \{\mathbf{0}, \mathbf{1}, \mathbf{3}\} \\
I(d_2) &= \{\mathbf{0}, \mathbf{2}\}
\end{aligned}$$

Then, the expansions of the two literal sets are:

$$\{a(\underline{0}), a(\underline{1}), a(\underline{3})\}; \quad \textit{and}$$
$$\{a(\underline{0}), a(\underline{2})\}$$

so the expansion of C is

$$E(C, D) = 1 \leq \{a(\underline{0}), a(\underline{1}), a(\underline{2}), a(\underline{3})\} \leq 2 \ .$$

The expansion of a rule is defined analogously to Definition 4.7.

Definition 4.9 *The* expansion *of a rule $r = C_0 \leftarrow C_1, \ldots, C_n$ with respect to a data model D is defined as follows:*

$$E(r, D) = E(C_0, D) \leftarrow E(C_1, D), \ldots, E(C_n, D) \ . \tag{16}$$

Example 4.12 *Continuing from Example 4.10, the first rule of* (13) *expands to:*

$$1 \leq \{hc(\underline{a}, \underline{b})\} \leq 1 \leftarrow 1 \leq \{\top\}$$

since there is only one pair in $I(edge)$ where **a** *occurs in the first argument. The corresponding rule for vertex c expands to:*

$$1 \leq \{hc(\underline{c}, \underline{a}), hc(\underline{c}, \underline{d})\} \leq 1 \leftarrow 1 \leq \{\top\} \ .$$

There is also one element **e** *in the universe of the example that is not a vertex. For it, the rule is expanded to:*

$$1 \leq \{\} \leq 1 \leftarrow 1 \leq \{\} \ .$$

Since the body is unsatisfiable, this rule has no effect at all and it may be dropped from the instantiated program.

The final rule of (5) *expands to:*

$$\leftarrow 1 \leq \{\text{not } r(\underline{a}), \text{not } r(\underline{b}), \text{not } r(\underline{c}), \text{not } r(\underline{d})\} \ .$$

Now we are ready to combine partial instantiation with expansion to define Herbrand instantiation **HI** for cardinality constraint rules and programs.

Definition 4.10 *Let $r = C_0 \leftarrow C_1, \ldots, C_n$ be a rule and D be a data model. Then, the* Herbrand instantiation **HI** *of r with respect to D is the set*

$$\mathbf{HI}(r, D) = \{E(r', D) \mid r' \in \mathit{inst}(r, D)\} \ . \tag{17}$$

Definition 4.11 *Let P be a cardinality constraint program and D be a data model. Then, the* Herbrand instantiation **HI** *of P with respect to D is the set*

$$\mathbf{HI}(P, D) = \bigcup_{r \in P} \mathbf{HI}(r, D) \ . \tag{18}$$

4.4 Reduct and Stable Model Semantics

Thus far we have only performed syntactic manipulation for a program P instantiating it with respect to a data model D. In this section we complete the semantics by defining a reduction for instantiated programs. The reduction is done with respect to a set M of ground atoms. Our presentation follows the one in [65].

We start by noting that a ground atom a is satisfied by M (denoted $M \models a$) if $a \in M$ and a ground negative literal $\text{not}(a)$ if $a \notin M$. Additionally, $M \models \top$ for all M.

Definition 4.12 *A set of ground atoms M* satisfies *an expanded constraint literal $C = L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \leq U$ if and only if $L \leq w(C, M) \leq U$ where*

$$w(C, M) = |\{a_i \mid a_i \in M\}| + |\{b_i \mid b_i \notin M\}| \tag{19}$$

is the number of basic literals in C that are satisfied by M.

An instantiated rule $r = C_0 \leftarrow C_1, \ldots, C_n$ is satisfied by S ($S \models r$) if and only if S satisfies C_0 whenever it satisfies each of $C_1, \ldots, C_n$. An instantiated program P is satisfied by S ($S \models P$) if S satisfies each rule in P.

Example 4.13 *Consider the expanded constraint literal C:*

$$2 \leq \{a, b, \text{not } c\} \leq 3$$

Now, $w(C, \{a\}) = 2$ so $\{a\} \models C$. On the other hand, $w(C, \{a, c\}) = 1$ so $\{a, c\} \not\models C$.

The reduct of a program consists of a set of Horn constraint rules. Since there are only positive literals and no upper bounds, the rules are monotonic [65] in the same way as ordinary Horn rules are; if $M \models C$, then $M' \models C$ for any set $M' \supseteq M$ so deducing new atoms can never render prior deductions false.

Definition 4.13 *The* deductive closure *$cl(P)$ of a set P of ground Horn constraint rules is the least fixed point of the operator T_P where T_P is defined as follows*

$$T_P(M) = \{a \mid a \leftarrow C_1, \ldots, C_n \in P \text{ and for all } i, M \models C_i\} \ . \tag{20}$$

The deductive closure of a set P of ground Horn constraint rules corresponds to the unique minimal model of P [65].

Example 4.14 *Let P be the set of Horn constraint rules:*

$$\begin{aligned} a &\leftarrow 1 \leq \{a\} \\ b &\leftarrow 0 \leq \{b\} \\ c &\leftarrow 2 \leq \{b, d\}, 1 \leq \{b, a\} \ . \end{aligned}$$

The deductive closure of P is $\{b\}$. However, if we add the rule

$$d \leftarrow 1 \leq \{a, b, c\}$$

to the set, then the closure is $\{b, d, c\}$.

Definition 4.14 *The* reduct C^M *of an expanded constraint literal* $C = L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \leq U$ *with respect to a set of atoms* M *is the constraint*

$$L' \leq \{a_1, \ldots, a_n\} \tag{21}$$

where the lower bound

$$L' = L - |\{b_i \mid b_i \notin M\}| \ . \tag{22}$$

In effect, all negative literals and the upper bound in the constraint are removed and the number of satisfied negative literals is subtracted from the lower bound.

Example 4.15 *Given a constraint* C

$$1 \leq \{a_1, \text{not } b_1, \text{not } b_2\} \leq 3 \ ,$$

its reduct with respect to $\{b_1\}$ *is the constraint*

$$C^{\{b_1\}} = 0 \leq \{a_1\}$$

and with respect to $\{b_1, b_2\}$

$$C^{\{b_1, b_2\}} = 1 \leq \{a_1\} \ .$$

Definition 4.15 *Let* P *be an instantiated cardinality constraint program and* M *a set of atoms. Then, the reduct* P^M *of* P *with respect to* M *is defined as follows:*

$$P^M = \{p \leftarrow C_1^M, \ldots, C_n^M \mid C_0 \leftarrow C_1, \ldots, C_n \in P, p \in C_0 \text{ and } M \models p, M \models C_1, \ldots, C_n\} \tag{23}$$

The main idea here is that the reduct of a single rule r with respect to M is a set of simple rules whose bodies are the reducts of the body of r and heads are those atoms of $head(r)$ that are true in M.

Definition 4.16 *A set* M *of atoms is a* stable model *of an instantiated program* P *if and only if the following two conditions hold:*

1. $M \models P$,
2. $M = cl(P^M)$

The first condition ensures that all rules are satisfied by the model and the second one ensures that all atoms in M occur in heads of rules in P and every atom in it has a non-circular justification.

Example 4.16 *Consider the instantiated program* P*:*

$$1 \leq \{a, c\} \leq 1 \leftarrow 1 \leq \{\text{not } b\}$$
$$b \leftarrow 2 \leq \{a, \text{not } c\} \ .$$

Now, P *has only one stable model,* $\{c\}$*. We immediately see that* $\{c\} \models P$*. The reduct* $P^{\{c\}}$ *is:*

$$c \leftarrow 0 \leq \{\}$$

whose closure is $\{c\}$*. Thus,* $\{c\}$ *is stable. On the other hand,* $\{a, c\}$ *is not a stable model since it does not satisfy the first rule, and likewise* $\{b\}$ *is not since* $cl(P^{\{b\}}) = \emptyset$*.*

Finally, we can put everything together and define the stable model semantics for cardinality constraint programs with variables.

Definition 4.17 *Given a cardinality constraint program P and a data model D, a set M of ground atoms is a stable model of $\langle P, D\rangle$ if and only if M is a stable model of* **HI**(P, D)*. We denote the set of all stable models of $\langle P, D\rangle$ by $\mathcal{A}(P, D)$.*

Next, we show that the stable model semantics for cardinality constraint programs is a true generalization of the stable model semantics for normal logic programs.

We say that a cardinality constraint program is *normal* if and only if all rules in it are of the form:

$$\begin{aligned} 1 \leq \{h : \top\} \leftarrow 1 \leq \{a_1 : \top\}, \ldots, 1 \leq \{a_n : \top\}, \\ 1 \leq \{\text{not } b_1 : \top\}, \ldots, 1 \leq \{\text{not } b_m : \top\} \ . \end{aligned} \tag{24}$$

A rule of this form corresponds directly to a normal rule

$$h \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m \ .$$

Theorem 4.1 *Let P be a normal cardinality constraint program. Then, M is a stable model of P if and only if M is a stable model of a normal logic program P_n that is obtained from P by replacing all constraint literals $1 \leq \{l : \top\}$ by the literal l.*

Proof. Let a rule r be of the form (24) and M be a set of atoms. Then, if $h \notin M$ the reduct r^M is empty, and if $h \in M$, the reduct is:

$$h \leftarrow 1 \leq \{a_1\}, \ldots, 1 \leq \{a_n\}, \ldots, l_1 \leq \{\}, \ldots, l_m \leq \{\} \tag{25}$$

where $l_i = 1$ if $b_i \in M$, and 0 otherwise. If $l_i = 1$ for any i, then r^M contains an unsatisfiable body literal and thus it is trivially true and cannot be used to justify h in the stable model. Otherwise, all literals $0 \leq \{\}$ are trivially true and can be left out from the rule.

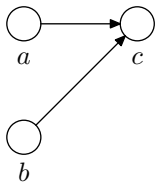
Let a rule r_n be the normal logic program version of r. Then, the reduct r_n^M is the rule:

$$h \leftarrow a_1, \ldots, a_n \tag{26}$$

if for all i, $b_i \notin M$ and empty otherwise. We immediately see that the rules (25) and (26) behave exactly the same under the corresponding T_P operators and they are satisfied by the same sets of atoms.

The only case when the reducts r^M and r_n^M are essentially different is when $h \notin M$ and all negative literals in the body are satisfied since then r^M is empty but r_n^M is not. Suppose that a_i are all true in $cl(P_n^M)$. Then, also $h \in cl(P_n^M)$ so $cl(P_n^M) \neq M$ and M is not a stable model. Thus, in any stable model M' of P_n, at least one a_i is not true and the body of (26) is not satisfied so it is not used to justify any atom in the model, and leaving it out completely from P^M does not change the minimal model of the reduct.

Thus, the sets of stable models of P and P_n are equal. □



$$
\begin{aligned}
D &= \langle U, \Sigma, I \rangle \\
U &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{red}, \mathbf{blue}\} \\
\Sigma &= \langle \{vtx/1, edge/2, color/1\}, \mathcal{N}, \mathcal{N} \rangle \\
\mathcal{N} &= \{\ \underline{a}, \underline{b}, \underline{c}, \underline{red}, \underline{blue}\ \} \\
I(vtx) &= \{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \\
I(edge) &= \{\langle \mathbf{a}, \mathbf{c} \rangle, \langle \mathbf{b}, \mathbf{c} \rangle\} \\
I(color) &= \{\mathbf{red}, \mathbf{blue}\} \\
I(\underline{a}) &= \mathbf{a}; I(\underline{b}) = \mathbf{b}; I(\underline{c}) = \mathbf{c}; I(\underline{red}) = \mathbf{red}; \\
I(\underline{blue}) &= \mathbf{blue}
\end{aligned}
$$

Figure 3: A sample graph for vertex coloring

4.5 Example

Now we look at a simple cardinality constraint program and go through all steps that are necessary in the definition of stable models. Let P be a program that finds graph n-colorings:

$$
\begin{aligned}
1 \leq \{\rho C.\langle \mathit{has\text{-}color}(V, C) : \mathit{color}(C) \rangle\} \leq 1 \leftarrow \mathit{vertex}(V) \\
\leftarrow \mathit{has\text{-}color}(X, C), \mathit{has\text{-}color}(Y, C), \mathit{edge}(X, Y), \mathit{color}(C)\ .
\end{aligned}
$$

For the sake of the example, let us find a 2-coloring for the graph shown in Figure 3. In the first step we remove all global variables by creating a partial instantiation. For the sake of clarity, we leave out rules with trivially unsatisfiable bodies as well as atoms whose predicate symbols are data predicates. The resulting instantiation is:

$$
\begin{aligned}
1 \leq \{\rho C.\langle \mathit{has\text{-}color}(\underline{a}, C) : \mathit{color}(C) \rangle\} \leq 1 \leftarrow \\
1 \leq \{\rho C.\langle \mathit{has\text{-}color}(\underline{b}, C) : \mathit{color}(C) \rangle\} \leq 1 \leftarrow \\
1 \leq \{\rho C.\langle \mathit{has\text{-}color}(\underline{c}, C) : \mathit{color}(C) \rangle\} \leq 1 \leftarrow \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{a}, \underline{red}), \mathit{has\text{-}color}(\underline{c}, \underline{red}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{b}, \underline{red}), \mathit{has\text{-}color}(\underline{c}, \underline{red}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{a}, \underline{blue}), \mathit{has\text{-}color}(\underline{c}, \underline{blue}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{b}, \underline{blue}), \mathit{has\text{-}color}(\underline{c}, \underline{blue})
\end{aligned}
$$

Next, we expand the three literal sets to get the Herbrand instantiation of $P_2 = \mathbf{HI}(P_1, D)$:

$$
\begin{aligned}
1 \leq \{\mathit{has\text{-}color}(\underline{a}, \underline{red}), \mathit{has\text{-}color}(\underline{a}, \underline{blue})\} \leq 1 \leftarrow \\
1 \leq \{\mathit{has\text{-}color}(\underline{b}, \underline{red}), \mathit{has\text{-}color}(\underline{b}, \underline{blue})\} \leq 1 \leftarrow \\
1 \leq \{\mathit{has\text{-}color}(\underline{c}, \underline{red}), \mathit{has\text{-}color}(\underline{c}, \underline{blue})\} \leq 1 \leftarrow \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{a}, \underline{red}), \mathit{has\text{-}color}(\underline{c}, \underline{red}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{b}, \underline{red}), \mathit{has\text{-}color}(\underline{c}, \underline{red}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{a}, \underline{blue}), \mathit{has\text{-}color}(\underline{c}, \underline{blue}) \\
\quad \leftarrow \mathit{has\text{-}color}(\underline{b}, \underline{blue}), \mathit{has\text{-}color}(\underline{c}, \underline{blue})
\end{aligned}
$$

Consider first the model candidate:

$$
M_1 = \{\mathit{has\text{-}color}(\underline{a}, \underline{red}), \mathit{has\text{-}color}(\underline{a}, \underline{blue})\}\ .
$$

We see that $M_1 \not\models P_2$, so M_1 has to be rejected. Next we try:

$$M_2 = \{has\text{-}color(\underline{a}, \underline{red}), has\text{-}color(\underline{b}, \underline{red}), has\text{-}color(\underline{c}, \underline{blue})$$

The reduct $P_2^{M_2}$ is

$$\begin{aligned}
has\text{-}color(\underline{a}, \underline{red}) &\leftarrow \\
has\text{-}color(\underline{b}, \underline{red}) &\leftarrow \\
has\text{-}color(\underline{c}, \underline{blue}) &\leftarrow \\
\leftarrow has\text{-}color(\underline{a}, \underline{red}), &has\text{-}color(\underline{c}, \underline{red}) \\
\leftarrow has\text{-}color(\underline{b}, \underline{red}), &has\text{-}color(\underline{c}, \underline{red}) \\
\leftarrow has\text{-}color(\underline{a}, \underline{blue}), &has\text{-}color(\underline{c}, \underline{blue}) \\
\leftarrow has\text{-}color(\underline{b}, \underline{blue}), &has\text{-}color(\underline{c}, \underline{blue})
\end{aligned}$$

The deductive closure is:

$$\begin{aligned}
cl(P_2^{M_2}) &= \{has\text{-}color(\underline{a}, \underline{red}), has\text{-}color(\underline{b}, \underline{red}), has\text{-}color(\underline{c}, \underline{blue})\} \\
&= M_2,
\end{aligned}$$

so M_2 is a stable model.

5 OMEGA-RESTRICTED PROGRAMS

In the previous section we defined the semantics of a program with global variables to coincide with the stable models of its instantiation. While this is theoretically a clean way to do the definition, there are several practical problems with it. Perhaps the most important consequence is that if we use the Herbrand interpretation for some function symbols, then the problem of finding a stable model becomes undecidable [37].

Thus, we are interested in finding a class of cardinality constraint programs for which we can guarantee that we can always compute its stable models. Moreover, we want that the class is syntactic so that we can easily check whether a program belongs to it or not.

Another practical viewpoint is that the size of the full Herbrand instantiation of even a function-free program may be exponential compared to the original program in the general case. A large number of rules that occur in the Herbrand instantiation are irrelevant since their bodies cannot be satisfied in any stable model and they can be left out of the instantiation without affecting the stable models.

One common strategy of reducing the size of the ground instantiation is to demand that all rules of the program are *range-restricted* in the sense that if a variable occurs in a rule, then it has to occur also in a positive literal in the rule body. Thus, the rule

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

is range-restricted but

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

is not since neither X nor Y occur in a positive literal in the rule body. This way it is possible to compute the set of rules with possibly satisfiable bodies by disregarding all negative literals and then computing the deductive closure of the rules starting from ground facts. This approach makes it possible to use deductive database techniques to further optimize the instantiation [23].

In this section we define a different form of range-restriction, namely the class of ω-restricted programs that were originally introduced for normal logic programs in [75]. Here we generalize the ω-stratification concept to handle the new syntactic extensions.

A major motivation behind ω-restricted programs is to allow the programmer to define a data model of a program using the same syntax as the rest of the program. We do it by allowing some program predicates that have fixed extensions to act as data predicates for the rest of the program. We call them *domain predicates* and they are defined using a stratifiable set of simple rules. Such a set of rules has a unique least model and we add it to the data model. Then we instantiate the rest of the program with respect to the augmented data model.

In practice we construct a stratification of predicate symbols such that a predicate p is on a at least as high level as a predicate q if q occurs in the body of a rule for p. The unique stable model of the domain predicates is then constructed one stratum a time, starting from the lowest one. The

stratification contains an extra level, ω-stratum, to hold all the predicates whose extensions cannot be syntactically guaranteed to be fixed.

In addition to domain predicates we may still use arbitrary data models. In practice, a data model defines some built-in predicates and functions such as equality ($=$) and arithmetic operators ($+$, $-$, ...).

Since we now allow some program predicates to work as data predicates, we also have to allow those predicates to occur in conditions of literal sets. Also, from now on we do not have to use $\top$ as a program predicate anymore so we can assign it to be solely a data predicate. We do this to simplify some of the following definitions.

We say that a constraint literal in a rule body is *simple* if it is of the form $1 \leq \{a : \top\}$ where a is an atom. Intuitively, a rule is ω-restricted if each variable that occurs in it occurs also in some positive simple constraint literal whose main predicate is on a strictly lower stratum than the head of the rule. A program is ω-restricted if and only if all its rules are. This condition ensures that even if there are some uninterpreted function symbols in a program and its Herbrand instantiation is infinite, all its stable models are still finite. Moreover, computing those models is decidable.

Example 5.1 *We will continue to use the Hamiltonian cycle problem as our running example. However, we now extend the program to make it work also for undirected graphs. We do this by explicitly forcing the $edge/2$ predicate to be symmetric by adding the rule:*

$$edge(Y, X) \leftarrow edge(X, Y), vtx(X), vtx(Y) \ .$$

Thus, the complete program is:

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

Also, we now use a data model without any data predicates at all. Instead, we define both vtx/1 and edge/2 as domain predicates. That is, they are program predicates with fixed extensions that act as data predicates for other predicates.

Our sample graph that was first presented in Figure 2 on page 15 can be encoded as the set of following ten facts:

$$\begin{array}{cccc}
vtx(a) \leftarrow & vtx(b) \leftarrow & vtx(c) \leftarrow & vtx(d) \leftarrow \\
edge(a,b) \leftarrow & edge(b,c) \leftarrow & edge(b,d) \leftarrow & edge(c,d) \leftarrow \\
edge(c,a) \leftarrow & edge(d,c) \leftarrow & . &
\end{array}$$

5.1 Dependency Graphs

In this section we define what it means for a predicate to depend on another. We start by defining some auxiliary notations. Given a rule r =

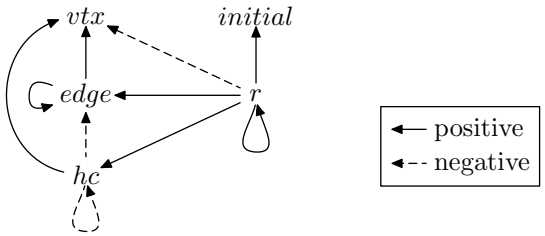


Figure 4: The dependency graph of the Hamiltonian cycle program.

$C_0 \leftarrow C_1, \ldots, C_n$, we use $head(r)$ to denote the set of main literals of literal sets that occur in C_0, that is $head(r) = \bigcup_{S \in C_0} lit(S)$.

We also use the notations $body^+(r)$ and $body^-(r)$ to denote the sets of positive and negative basic literals that occur somewhere in the body of a rule r, either as a main literal or in some condition, and whose predicate symbols are program predicates. Similarly, we use $body_s(r)$ to denote the set of basic literals that occur in simple constraint literals of the form $1 \leq \{a : \top\}$ somewhere in the rule body.

Definition 5.1 *Let P be a cardinality constraint program. Then, the* one-step dependency relation $\mathfrak{D}_1(P) \subseteq \mathcal{P}_P(P) \times \mathcal{P}_P(P)$ *is defined as follows:*

$$\begin{aligned}
\mathfrak{D}_1^+(P) = \{\langle pred(h), pred(l)\rangle \mid{}& \exists r \in P : \\
& h \in head(r) \wedge l \in body^+(r)\} \qquad (27)\\
\mathfrak{D}_1^-(P) = \{\langle pred(h), pred(l)\rangle \mid{}& \exists r \in P : \\
& h \in head(r) \wedge l \in body^-(r)\} \\
\cup\{\langle pred(l), pred(a)\rangle \mid{}& \exists S \in \mathcal{L}(P) : \qquad (28)\\
& l = lit(S) \wedge a \in conds(S)\} \\
\cup\{\langle pred(p), pred(p)\rangle \mid{}& \exists r \in P : \\
& p \in head(r) \text{ and } r \text{ is not simple}\} \\
\mathfrak{D}_1(P) = \mathfrak{D}_1^+(P) \cup \mathfrak{D}_1^-(P) & \qquad (29)
\end{aligned}$$

Note that we define the dependency relation only over program predicates. The reason for this that otherwise we could not guarantee that a finite ω-restricted program has a finite stable model, since an infinite data model might force the models to be also infinite. This point is further discussed in Section 8. Another note is that since we defined $\mathcal{P}_P(P)$ to be finite, also $\mathfrak{D}_1(P)$ is finite even if P is infinite.

The definition of $\mathfrak{D}_1^-(P)$ may seem unnecessarily complex. Again, we defer explaining the motivation behind it to Section 5.2 where we give the formal definition of stable models for ω-restricted programs.

The one-step dependency relation may be drawn as a graph. For example, the dependency graph of the program in Example 5.1 is shown in Figure 4.

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 edges between p and q is negative, then p depends negatively on q.

Definition 5.2 *A* dependency path π_P *of a logic program* P *is a sequence*

$$\pi_P = \langle p_1, p_2, \ldots, p_n \rangle \tag{30}$$

where $p_i \in \mathcal{P}_P(P)$ *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* $\overline{\pi}_P$*) 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* $\overline{\Pi}_P$.

Definition 5.3 *The* dependency relation $\mathfrak{D}(P) \subseteq \mathcal{P}_P(P) \times \mathcal{P}_P(P)$ *of a logic program* P *is defined as follows:*

1. $\mathfrak{D}(P) = \{\langle p, q \rangle \mid \exists \pi \in \Pi_P : \pi = \langle p, \ldots, q \rangle\}$;
2. $\mathfrak{D}^-(P) = \{\langle p, q \rangle \mid \exists \overline{\pi} \in \overline{\Pi}_P : \overline{\pi} = \langle p, \ldots, q \rangle\}$; *and*
3. $\mathfrak{D}^+(P) = \mathfrak{D}(P) \setminus \mathfrak{D}^-(P)$.

Next, we define the concept of ω-stratification. The definition extends the usual definition of stratification [2] by adding a new stratum, the ω-stratum, for the predicates that depend negatively on each other.

Definition 5.4 *An* ω-stratification *of a program* P *is a function* $\mathcal{S} : \mathcal{P}_P(P) \to \mathbb{N} \cup \{\omega\}$ *such that:*

1. $\forall p_1 \forall p_2 (\langle p_1, p_2 \rangle \in \mathfrak{D}^+(P) \Rightarrow \mathcal{S}(p_1) \geq \mathcal{S}(p_2))$; *and*
2. $\forall p_1 \forall p_2 (\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P) \Rightarrow \mathcal{S}(p_1) > \mathcal{S}(p_2) \vee \mathcal{S}(p_1) = \omega)$.

A stratification $\mathcal{S}$ *is* strict *if*

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* $p_1 \in \mathcal{P}_P(P)$ *it holds that if* $\mathcal{S}(p_1) = \omega$, *then there exists a predicate* $p_2 \in \mathcal{P}_P(P)$ *such that* $\mathcal{S}(p_2) = \omega$ *and* $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$.

We use the convention that $\omega > n$ 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 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.

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 to 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.

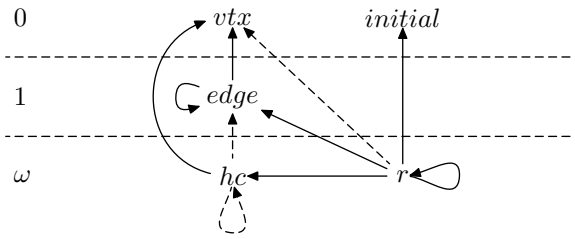


Figure 5: A stratification of the Hamiltonian cycle program

Example 5.2 *Consider Example 5.1. We can construct a strict ω-stratification $\mathcal{S}$ for the program by looking at its dependency graph. As there are no edges 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 5.3 *Consider the program P:*

$$odd(X+1) \leftarrow even(X), number(X)$$
$$even(X+1) \leftarrow odd(X), number(X)$$

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

As the set of program predicates $\mathcal{P}_P(P)$ is finite, the following proposition immediately follows:

Proposition 5.1 *The number of non-empty strata in an ω-stratification of a cardinality constraint program is finite.*

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

Definition 5.5 *The ω-valuation of a rule r under an ω-stratification $\mathcal{S}$ is the function:*

$$\Omega(r, \mathcal{S}) = \max(\{\mathcal{S}(pred(l)) \mid l \in head(r)\}) \tag{31}$$

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

$$\Omega(V, r, \mathcal{S}) = \min(\{\mathcal{S}(pred(a)) \mid a \in body_s(r) \wedge V \in vars(a)\} \cup \{\omega\}) \tag{32}$$

Example 5.4 *Let $\mathcal{S}$ be as defined in Example 5.2. Consider the rule r:*

$$1 \leq \{\rho Y.\langle hc(X,Y) : edge(X,Y)\rangle\} \leq 1 \leftarrow vtx(X)$$

Now

$$\Omega(r, \mathcal{S}) = \mathcal{S}(hc) = \omega$$
$$\Omega(X, r, \mathcal{S}) = \min\{\mathcal{S}(vtx), \omega\} = 0$$

Definition 5.6 *A literal set $\rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m \rangle$ is* ω-restricted under a stratification $\mathcal{S}$ *if and only if $\mathcal{S}(pred(l)) > \mathcal{S}(pred(a_i))$ for all $i \in [1, m]$ and $\{X_1, \ldots, X_n\} \subseteq \bigcup_{i \in [1,n]} vars(a_i)$.*

In practice, this definition means that the main literal has to be on a strictly higher stratum than the conditions, and that all local variables have to occur in conditions.

A rule is ω-restricted if all literal sets in it are restricted and if all global variables that occur in it also occur 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 5.7 *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*

$$\forall V \in vars(r) : \Omega(V, r, \mathcal{S}) < \Omega(r, \mathcal{S}) \ .$$

and all literal sets $\mathcal{L}(r)$ are ω-restricted under $\mathcal{S}$.

Example 5.5 *Consider the rule r:*

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

This rule is not ω-restricted since $\forall \mathcal{S} : \Omega(r, \mathcal{S}) = \Omega(X, r, \mathcal{S})$.

Finally, 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 5.8 *Let P be an ω-restricted program. Then a predicate $p \in \mathcal{P}_P(P)$ is a* domain predicate *if and only if there exists a strict ω-stratification $\mathcal{S}$ such that $\mathcal{S}(p) < \omega$.*

We will use the term *domain literal* to denote a simple constraint literal whose predicate is a domain predicate. We note here that if a predicate is on a finite stratum in one strict stratification, then it is on a finite one in each such stratification, since a predicate may be on the ω-stratum only if it depends on some predicate that depends on itself negatively. Also, it turns out that every cardinality constraint program has a strict ω-stratification. We devote rest of this section to showing this.

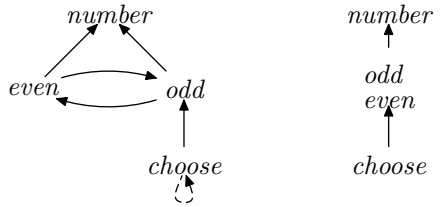
We can create a strict ω-stratification of a program P using the algorithm presented in Figure 7. Next, we explain what that algorithm does in practice.

Definition 5.9 *Let P be a cardinality constraint program. Then, its* strongly connected component graph $\mathrm{SCC}(P) = \langle V_P, E_P, N_P \rangle$ *is defined as follows:*

1. *$v \in V_P$ if and only if the following three conditions hold:*
   (a) *$v \subseteq \mathcal{P}_P(P)$;*
   (b) *$\forall x, y \in v : \langle x, y \rangle \in \mathfrak{D}(P) \wedge \langle y, x \rangle \in \mathfrak{D}(P)$; and*
   (c) *$\forall x, y \in \mathcal{P}_P(P) : x \in v \wedge \langle x, y \rangle \in \mathfrak{D}(P) \wedge \langle y, x \rangle \in \mathfrak{D}(P)$ implies that $y \in v$.*

$$even(X+1) \leftarrow odd(X), number(X)$$
$$odd(X+1) \leftarrow even(X), number(X)$$
$$\{choose(X)\} \leftarrow odd(X)$$

(a) Program



(b) Dependency graph (c) SCC graph

Figure 6: An example of SCC graph formation.

2. $\langle v_1, v_2 \rangle \in E_P$ *iff* $\exists x \in v_1 : \exists y \in v_2 : x \neq y$ *and* $\langle x, y \rangle \in \mathfrak{D}(P)$.

3. $N_P \subseteq V_P$ *such that*

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

In effect, the nodes of SCC(P) correspond with the strongly connected components of the dependency graph of P. The set N_P contains all those strongly connected components that contain predicates that depend negatively on itself.

The basic intuition of the algorithm *create_stratification* (Figure 7) is that it first marks all components in N_P as belonging to the ω-stratum, and then it computes a depth-first search on 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 the maximum s of their strata is computed recursively, and all predicates in v are assigned to the stratum $s+1$ where we suppose that $\omega + 1 = \omega$. In Figure 8 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.

Proposition 5.2 *Let P be a cardinality constraint program. Then, the result $\mathcal{S}$ computed by the algorithm create_stratification is a strict ω-stratification of P.*

Proof. Suppose that for predicates p_1 and p_2 it holds that $\langle p_1, p_2 \rangle \in \mathfrak{D}^+(P)$. Then there are two further 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 edge 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 $\mathcal{S}(p_2) = \omega$.

Next, consider the case where $\langle p_1, p_2 \rangle \in \mathfrak{D}^-(P)$. Then, if $\langle p_2, p_1 \rangle \notin \mathfrak{D}(P)$, *find_stratum* behaves as above and either $\mathcal{S}(p_1) > \mathcal{S}(p_2)$ 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

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**
        $s' :=$ *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 7: Creating an ω-stratification

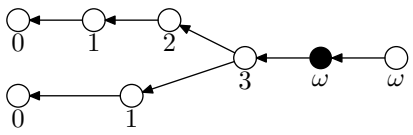


Figure 8: A sample SCC graph and its stratification

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. □

Corollary 5.1 *Let P be a cardinality constraint program. Then, there exists a strict stratification $\mathcal{S}$ of P.*

Theorem 5.1 *The problem of deciding whether a finite cardinality constraint program is ω-restricted is decidable in a polynomial time.*

Proof. The SCC graph of a cardinality constraint program P can be constructed in a linear time using, for example, the well-known Tarjan's algo-

rithm [61, pp. 481–483]. After that the algorithm *create_stratification* creates a strict ω-stratification $\mathcal{S}$ for P using a quadratic amount of time.[5] After that, we can check that each rule of P is ω-restricted in a linear time.

Furthermore, suppose that P is not ω-restricted under $\mathcal{S}$ but there is another strict ω-stratification $\mathcal{S}'$ such that P is ω-restricted under it. Then there exists some rule r and some variable V where $\Omega(V, r, \mathcal{S}) \geq \Omega(r, \mathcal{S})$ but $\Omega(V, r, \mathcal{S}') < \Omega(r, \mathcal{S}')$. The first condition implies that for literals $l \in body_s(r)$ such that $V \in vars(l)$, $\mathcal{S}(pred(l)) \geq \Omega(r, \mathcal{S})$. Since for all $h \in head(r)$ and $l \in body_s(r)$ it holds that $\langle pred(h), pred(l)\rangle \in \mathfrak{D}(P)$, we see that $\Omega(V, r, \mathcal{S}) = \Omega(r, \mathcal{S})$ and all h and l either belong to the same SCC or are belong to the ω-stratum since else the algorithm *create_stratification* would have assigned the heads on a strictly higher stratum. However, this causes contradiction with the assumption that $\Omega(V, r, \mathcal{S}') < \Omega(r, \mathcal{S}')$ as a strict stratification may not assign a predicate p into a lower stratum than a predicate q if $\langle p, q\rangle \in \mathfrak{D}$. □

5.2 The Stable Model Semantics of ω-Restricted Programs

The stable model semantics of an ω-restricted program P is defined by splitting P into n smaller programs where n is the number of different strata in its stratification. A stable model is constructed incrementally, starting from rules for predicates on 0-stratum, continuing upwards until the ω-stratum is reached, if necessary.

Definition 5.10 *Let P be a cardinality constraint program and $\mathcal{S}$ be its strict ω-stratification such that P is ω-restricted under $\mathcal{S}$. Then, the* stratum program $P_i^{\mathcal{S}}$ *is defined as follows:*

$$P_i^{\mathcal{S}} = \{r \in P \mid \Omega(r, \mathcal{S}) = i\} \ . \tag{33}$$

The data model of the program is also constructed incrementally, with each strata progam instantiated with respect to the stable model of the previous strata.

As all rules belonging to finite strata are simple, their instantiations are of the form:

$$1 \leq \{h\} \leftarrow C_1, \ldots, C_n$$

where h is an atom and all constraint literals C_i are of the form:

$$L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}$$

where all atoms b_i belong to a strictly lower stratum than the head h and their total extensions are known at the instantiation time. Thus, for each atom b_i we know whether $\text{not}(b_i)$ is true or false, and we can replace it by its truth value. If we also replace the head $1 \leq \{h\}$ by the corresponding atom h, the resulting rules will be Horn constraint rules and thus we can find their deductive closure using the cl operator.

Formally, we alter the definitions of expansions of literal sets, constraint literals, and rules so that data predicates that occur as main literals in literal

[5] The algorithm can be modified to achieve a linear time bound by caching results of *find_stratum*.

sets are evaluated. Then, the Herbrand instantiation for simple rules is defined in the same way as in the previous section but the new definitions for expansions are used instead.

Definition 5.11 *The* simple expansion *of a literal set* $S = \rho X_1 \cdots X_n.\langle l : a_1, \ldots, a_m \rangle$ *with respect to a data model* D *is the set:*

$$E'(S, D) = \{l\sigma \mid l \notin \mathcal{P}_D \wedge \sigma \in subs(vars_{\mathfrak{L}}(S), D) \\ \wedge \forall i \in [1, m] : D \models a_i\sigma\} \ . \tag{34}$$

The satisfied expansion *of* S *with respect to* D *is the set:*

$$E'_s(S, D) = \{l\sigma \mid l \in \mathcal{P}_D \wedge \sigma \in subs(vars_{\mathfrak{L}}(S), D) \wedge D \models l\sigma \\ \wedge \forall i \in [1, m] : D \models a_i\sigma\} \ . \tag{35}$$

Definition 5.12 *The* simple expansion *of a constraint literal* $C = L \leq \{S_1, \ldots, S_n\} \leq U$ *with respect to a data model* D *is defined as follows:*

$$E'(C, D) = L' \leq E'(S_1, D) \cup \cdots \cup E'(S_n, D) \leq U \tag{36}$$

where

$$L' = L - |E'_s(S_1, D) \cup \cdots \cup E'_s(S_n, D)| \ . \tag{37}$$

Definition 5.13 *The* simple expansion *of a simple rule* $r = 1 \leq \{h : \top\} \leftarrow C_1, \ldots, C_n$ *with respect to a data model* D *is:*

$$E'(r, D) = h \leftarrow E'(C_1, D), \ldots, E'(C_n, D) \ .$$

Example 5.6 *Let* $C = 3 \leq \{\rho X.\langle a(X) : d(X)\rangle, \rho Y.\langle \text{not } b(Y) : d(Y)\rangle\}$ *be a constraint literal where* $d/1, b/1 \in \mathcal{P}_D$ *and* $a/1 \notin \mathcal{P}_D$*. Suppose further that* D *is defined so that the extension of* $d/1$ *is* $\{1, 2\}$ *and the extension of* $b/1$ *is* $\{1\}$*. Let* $S_1 = \rho X.\langle a(X) : d(X)\rangle$ *and* $S_2 = \rho Y.\langle \text{not } b(Y) : d(Y)\rangle$*. Then,*

$$\begin{aligned} E'(S_1, D) &= \{a(1), a(2)\} \\ E'_s(S_1, D) &= \emptyset \\ E'(S_2, D) &= \emptyset \\ E'_s(S_2, D) &= \{\text{not } b(2)\} \ . \end{aligned}$$

Thus, the expansion of C *is:*

$$E'(C, D) = 2 \leq \{a(1), a(2)\} \ .$$

Definition 5.14 *Let* $r = C_0 \leftarrow C_1, \ldots, C_n$ *be a simple rule and* D *a data model. Then, its* Herbrand instantiation *is the set:*

$$\mathbf{HI}'(r, D) = \{E'(r', D) \mid r' \in inst(r, D)\} \ . \tag{38}$$

Let P *be a set of simple rules. Then:*

$$\mathbf{HI}'(P, D) = \bigcup_{r \in P} \mathbf{HI}'(r, D) \ .$$

We start the construction of the full data model from an arbitrary data model D and augment it by extensions of domain predicates.

Definition 5.15 *Let $D = \langle U, \langle \mathcal{P}_D, \mathcal{F} \rangle, I \rangle$ be a data model and M be a set of ground atoms. Then D* augmented *by M (denoted $D \uplus M$) is the data model $D' = \langle U, \langle \mathcal{P}'_D, \mathcal{F} \rangle, I' \rangle$ where*

$$\mathcal{P}'_D = \mathcal{P}_D \cup \{p \mid \ p \text{ is a predicate occurring in } M\}$$

and for all $p \in \mathcal{P}'_D$:

$$I'(p) = \begin{cases} I(p) \cup \{\langle I(t_1), \ldots, I(t_n)\rangle \mid p(t_1, \ldots, t_n) \in M\}, & \text{if } p \in \mathcal{P}_D \\ \{\langle I(t_1), \ldots, I(t_n)\rangle \mid p(t_1, \ldots, t_n) \in M\}, & \text{if } p \notin \mathcal{P}_D \ . \end{cases}$$

Let M_1, ..., M_n be sets of ground atoms. Then,

$$D \uplus \{M_i\}_{i=0}^{n} = D \uplus M_1 \uplus \cdots \uplus M_n$$

Definition 5.16 *Let P be a cardinality constraint program, $\mathcal{S}$ be its strict ω-stratification, and D be a data model. Then, $M_0^{\mathcal{S}} = cl(\mathbf{HI}'(P_0^{\mathcal{S}}, D))$, $D_0^{\mathcal{S}} = D \uplus M_0$, and for each $i \in \mathbb{N}$ we define recursively:*

1. $M_{i+1}^{\mathcal{S}} = cl(\mathbf{HI}'(P_{i+1}^{\mathcal{S}}, D_i^{\mathcal{S}}))$
2. $D_{i+1}^{\mathcal{S}} = D_i^{\mathcal{S}} \uplus M_{i+1}^{\mathcal{S}}$

The data model *of P is the set $D_P^{\mathcal{S}} = D \uplus \{M_i\}_{i=0}^{\infty}$.*

Note that by Proposition 5.1 only a finite number of strata are not empty, so in practice we can compute $D_P^{\mathcal{S}}$ in a finite number of steps.

Proposition 5.3 *Let P be a cardinality constraint program and $\mathcal{S}_1$ and $\mathcal{S}_2$ be two different strict ω-stratifications such that P is ω-restricted under both. Then, $D_P^{\mathcal{S}_1} = D_P^{\mathcal{S}_2}$.*

Proof. Suppose that $D_P^{\mathcal{S}_1} \neq D_P^{\mathcal{S}_2}$. Then, there is a ground atom a on which $D_P^{\mathcal{S}_1}$ and $D_P^{\mathcal{S}_2}$ disagree and which has the lowest ranking in both S_1 and S_2, This means that if $\min\{\mathcal{S}_1(pred(a)), \mathcal{S}_2(pred(a))\} = k$, then $D_{k-1}^{\mathcal{S}_1} = D_{k-1}^{\mathcal{S}_2}$. Let $\mathcal{S}$ denote the stratification for which $D_k^{\mathcal{S}} \models a$ and $\overline{\mathcal{S}}$ denote the one with $D_k^{\overline{\mathcal{S}}} \not\models a$. If there are more than one such atom, then let a be the one with the shortest derivation where the length of a derivation is defined to be the number of times the T operator has to applied before we know that a is in the minimal model of $\mathbf{HI}'(P_k^{\mathcal{S}}, D_{k-1}^{\mathcal{S}})$.

Now, there exists a rule $a \leftarrow body$ in the instantiation of $P_k^{\mathcal{S}}$ such that every constraint literal in *body* is of the form:

$$L \leq \{l_1, \ldots, l_n\}$$

and at least L of the basic literals l_i are either satisfied in $D_{k-1}^{\mathcal{S}}$ or have shorter derivations in $\mathbf{HI}'(P_k^{\mathcal{S}}, D_{k-1}^{\mathcal{S}})$ than a. In particular, if l_i is negative, then it has to belong to a lower stratum than a.

The same rule occurs also in the instantiation of $P^{\overline{\mathcal{S}}}_{k'}$ (where $\overline{\mathcal{S}}(pred(a)) = k' < \omega$) but $D^{\overline{\mathcal{S}}}_{k'} \not\models body$. Thus, *body* contains some ground literal l_i such that $D^{\mathcal{S}}_P \models l_i$ but $D^{\overline{\mathcal{S}}}_P \not\models l_i$. However, this contradicts our assumptions since l_i is either on a lower stratum than a or it has a shorter derivation than a. Thus, $D^{\mathcal{S}_1}_P = D^{\mathcal{S}_2}_P$. □

Because of the above proposition, we can simply leave the stratification out of the definition of $P^{\mathcal{S}}_i$ and $D^{\mathcal{S}}_P$ and simply write P_i and D_P.

Now we are ready to define stable models of ω-restricted programs.

Definition 5.17 *Let P be an ω-restricted program and D a data model. Then, M is a* stable model *of P if and only if M is a stable model of $\langle P_\omega, D_P \rangle$.*

Proposition 5.4 *For each $i \in \mathbb{N}$, M_i is the unique stable model of $\langle P_i, D \uplus \{M_i\}^{i-1}_{j=0} \rangle$.*

Proof. All rules on stratum 0 are ground facts of the form $a \leftarrow$. Thus, $cl(P_0)$ is trivially the unique stable model of $\langle P_0, D \rangle$. Suppose that there exists $k \in \mathbb{N}$ such that claim holds for all $i \leq k$. Now, consider stratum $k+1$. Either P_{k+1} is empty and $\emptyset$ is the unique stable model of $\langle P_{k+1}, D \uplus \{M_j\}^{k-1}_{j=0} \rangle$; or P_{k+1} consists of monotonic simple rules and $cl(\mathbf{HI}'(P_{k+1}, D \uplus \{M_j\}^{k-1}_{j=0}))$ is the unique stable model. □

We can now see the motivation behind the definition of $\mathfrak{D}^-(P)$ in (28). There we added a negative dependency between a main literal l and its conditions a_i in a set literal. In practice, this forces l to be on a strictly higher stratum than any of a_i, so the full extensions of a_i are known before the set literal is expanded.

Predicates that occur in heads of non-simple rules depend negatively on themselves because intuitively a rule of the form: $\{a\} \leftarrow$ introduces a choice where we can make a either true or false, so its extension is not fixed and we cannot use it as a domain predicate. However, there are also some non-simple rules, for example, $2 \leq \{a, b\} \leq 2 \leftarrow$, that would behave in a similar fashion than simple rules. Since it is not trivial to identify these rules in every situation and they can be rewritten as a set of simple rules, we chose to allow only simple rules for domain predicates.

In the Herbrand instantiation there may be many rules that do not affect the computation of stable models since their bodies are necessarily false. In particular, if a ground rule r has a domain atom a in its body where $D_P \not\models a$, it may never justify any atom in a stable model. We define that a rule is relevant if all domain literals in it are satisfied.

Definition 5.18 *Let R be a set of ω-restricted rules and D be a data model. Then,* relevant instantiation $\mathbf{HI}_r(R, D)$ *is the set:*

$$\mathbf{HI}_r(R, D) = \{E(r, D) \mid r \in inst(R, D) \text{ and } D \models C_i \text{ for all positive domain literals } C_i \text{ in the body of } r\} \quad . \tag{39}$$

Similarily, we use $\mathbf{HI}'_r(P, D)$ to denote the analogous relevant instantiation of simple rules.

When we compare this definition to the Definition 4.10, we see that $\mathbf{HI}_r(P, D) \subseteq \mathbf{HI}(P, D)$.

Theorem 5.2 *Let P be an ω-restricted program and D a data model. Then,* $\mathbf{HI}(P_\omega, D_P)$ *and* $\mathbf{HI}_r(P_\omega, D_P)$ *have the same stable models.*

Proof. To make the proof more legible, we use the shorthand notations $P_G = \mathbf{HI}(P_\omega, D_P)$ and $P_r = \mathbf{HI}_r(P_\omega, D_P)$.

Let M be a stable model of P_G. As $P_r \subseteq P_G$, we see that M is a model of P_r. Suppose that M is not stable with respect to P_r. Then there exists an atom $a \in M$ such that $a \notin cl(P_r^M)$ but $a \in cl(P_G^M)$. As $a \in cl(P_G^M)$, there is a rule $r = a \leftarrow body \in P_G^M$ such that $M \models body$. The rule r is a reduct of a rule $r' = h \leftarrow body'$ where $M \models body'$. However, since M is a model of P_r, this implies that $r' \notin P_r$. Since by definition $P_G \setminus P_r$ contains only rules with unsatisfiable bodies, we have $M \not\models body'$, a contradiction. Thus, M is a stable model of $\mathbf{HI}_r(P_\omega, D_P)$.

Next, let M be a stable model of P_r. As $P_G \setminus P_r$ contains only rules with unsatisfiable bodies, M is a model of P_G. Moreover, it is stable since no rule in $(P_G \setminus P_r)^M$ can contribute any new atoms to $cl(P_G^M)$. □

6 GENERALIZING CARDINALITY CONSTRAINTS

Thus far, we have examined the case where the bounds for constraint literals are integers and the satisfaction criterion is the sum of satisfied basic literals inside the constraints. However, we can also consider some other ways to define the satisfaction relation. We do this by defining general valuation functions that check whether a set of literals is satisfied. This approach was originally inspired by set constraint programs of Marek and Remmel [42].

6.1 Cardinality Constraints

A cardinality constraint consists of a set of literal sets and two integral bounds. The two bounds can be seen as attributes that are associated with the literal sets. We now generalize the notation by allowing a constraint literal to have a number of attributes.

Definition 6.1 *A* constraint literal signature $\mathfrak{C}$ *is a set of pairs* $\langle t, n\rangle$ *where* t *is a* type identifier *and* $n \in \mathbb{N}$ *the* arity.

Each constraint literal C that occurs in a program belongs to one of the types defined in the signature $\mathfrak{C}$ and has n terms associated with it.

Definition 6.2 *A* constraint literal *is a triple* $C = \langle t, S, A\rangle$ *where*

- t *is a type identifier;*
- $S = \langle S_1, \ldots, S_m\rangle$ *is a sequence of literal sets; and*
- $A = \langle t_1, \ldots t_n\rangle$ *a sequence of terms.*

A constraint literal belongs to the signature $\mathfrak{C}$ *if there exists a pair* $\langle t, n\rangle \in \mathfrak{C}$.

Here we define S as a sequence instead of a set since we might want to add attributes directly to the literal sets so we need a way that attributes are correctly associated. However, we will continue using the set notation in cases where the order is not significant.

Example 6.1 *The cardinality constraints have the signature* $\langle \mathrm{card}, 2\rangle$ *and an individual cardinality constraint* $L \leq \{a(X) : b(X)\} \leq U$ *is then expressed as* $\langle \mathrm{card}, \{a(X) : b(X)\}, \langle L, U\rangle\rangle$

For most of the time we will consider only ground programs in this section. In particular this means that all literal sets that we use will be of the form $l : \top$ where l is a ground basic literal and that we can treat all functions as constants. We use the notation *Lits*$(\mathcal{P}, \mathfrak{C})$ to denote the set of all constraint literals that can be constructed using the predicate signature $\mathcal{P}$ and constraint literal signature $\mathfrak{C}$, and *Rules*$(\mathcal{P}, \mathfrak{C})$ to denote all possible ground rules. A *formula* is either a basic literal, a constraint literal, or a rule. The set of all formulas that occur in a program P is denoted by $\mathfrak{F}(P)$ and the set of all atoms occurring in it by *At*(P).

6.2 Interpretations, Valuators, and Models

An interpretation is a function that assigns a truth value from a finite set $\mathcal{T}$ for each atom in $At(P)$. For most of the time we will use the set $\mathcal{T}_B = \{T, F\}$ containing the classical truth values true and false, but in Section 6.8 we add a third value: undefined U. For each set of truth values we also associate a partial order $\leq$ such that the partially ordered set (poset) $\langle \mathcal{T}, \leq \rangle$ is a lattice.

Definition 6.3 *A poset $\langle S, \leq; \vee, \wedge \rangle$ is a* lattice *if for all elements $x, y \in S$ the following two conditions hold:*

1. *there exists a unique least upper bound $z = (x \vee y) \in S$ such that for all z': $x \leq z' \wedge y \leq z'$ implies that $z \leq z'$; and*
2. *there exists a unique greatest lower bound $z = (x \wedge y) \in S$ such that for all z': $z' \leq x \wedge z' \leq y$ implies that $z' \leq z$.*

A lattice is complete *if for all subsets $S' \subseteq S$, the elements $\bigvee S'$ and $\bigwedge S'$ exist.*

Since $\langle \mathcal{T}, \leq \rangle$ is finite, there exist a unique minimum (denoted also by $\bot$) and a maximum (denoted $\top$) truth value [14]. If $x \leq y$ and $x \neq y$, we write $x < y$.

Example 6.2 *For the boolean truth values $\mathcal{T}_B = \{F, T\}$ we have $F < T$.*

Definition 6.4 *Given a ground program P and a set of truth values $\mathcal{T}$, an* interpretation *I is a function:*

$$I : At(P) \to \mathcal{T} \tag{40}$$

that assigns a truth value $I(A)$ for each atom $A \in At(P)$. The set of all interpretations of P using truth values $\mathcal{T}$ is denoted by $\mathfrak{I}(P, \mathcal{T})$.

We will use the symbol $\bot$ to also denote the interpretation that assigns the minimum truth value $\bot$ for all atoms when there is no risk of confusion.

A valuation extends an interpretation to cover all formulas of $\mathfrak{F}(P)$. As the way how we create a valuation depends on the exact semantics used, we define the concept of a valuator function. A valuator is a function that takes as its argument a formula and an interpretation and returns the valuation of the formula.

Definition 6.5 *Given a ground program P and a set $\mathcal{T}$ of truth values, a* valuator *$\mathfrak{v}$ is a function:*

$$\mathfrak{v} : \mathfrak{F}(P) \times \mathfrak{I}(P, \mathcal{T}) \to \mathcal{T} \tag{41}$$

that associates a truth value $\mathfrak{v}(f, I)$ for a formula f under an interpretation I such that $\mathfrak{v}(A, I) = I(A)$ for all $A \in At(P)$.

Example 6.3 *A valuator $\mathfrak{v}_C$ for standard expanded cardinality constraint literals $C = \langle \text{card}, \{l_1, \ldots, l_n\}, \langle L, U\rangle\rangle$ can be defined as follows:*

$$\mathfrak{v}_C(C, I) = \begin{cases} T, & \text{if } L \leq |\{l_i \mid 1 \leq i \leq n \wedge I(l_i) = T\}| \leq U \\ F, & \text{otherwise} \ . \end{cases}$$

We also need the valuator to assign a truth value to each rule $r = C_0 \leftarrow C_1, \ldots, C_n$. This can be done as follows:

$$\mathfrak{v}_C(r, I) = \begin{cases} T, & \text{if } \mathfrak{v}_C(C_0, I) = T \text{ or } \mathfrak{v}_C(C_i, I) = F \text{ for some } i \in [1, n] \\ F, & \text{otherwise} \ . \end{cases}$$

When we defined the stable model semantics in Definition 4.16 we had two conditions that a candidate model had to fulfill: each rule of the program had to be satisfied and the deductive closure of the reduct had to be the same as the model candidate. The rule satisfaction condition is simple in all two-valued logics as the rule is either true or false. However, when there are more truth values it is no longer clear what a satisfied rule is. Because of this we also have to define a two-valued version of a valuator that tells what truth values are acceptable for rules. For example, when we define the partial stable models in Section 6.8, a rule may be either true or undefined in a partial model, but it may not be false.

Definition 6.6 *Let P be a ground program, and $\mathfrak{v}$ be a valuator. Then a* $\mathfrak{v}$-satisfier $\mathfrak{s}_\mathfrak{v}$ *is a function:*

$$\mathfrak{s}_\mathfrak{v} : \mathfrak{F}(P) \times \mathfrak{I}(P, \mathcal{T}) \to \mathcal{T}_B \ . \tag{42}$$

We use the notation $I \models_\mathfrak{v} f$ to denote the case where $\mathfrak{s}_\mathfrak{v}(f, I) = T$. An interpretation I is a $\mathfrak{v}$-model *of P if for all rules $r \in P$, $I \models_\mathfrak{v} r$. The set of all* $\mathfrak{v}$*-models of P is denoted by $\mathcal{A}_\mathfrak{v}(P)$.*

We can say that a satisfier defines a semantics. That is, given a program P, a satisfier $\mathfrak{s}_\mathfrak{v}$ assigns a set of models $\{M \mid M \models_\mathfrak{v} P\}$ to the program.

6.3 Imposing an Order over Models

When we have an ordered set of truth values $\langle \mathcal{T}, \leq\rangle$, we can extend the order to cover the set $\mathfrak{I}(P, \mathcal{T})$ for a given program P so that $I_1 \leq I_2$ if all atoms have at least as high truth value in I_2 as in I_1. The reason why we want to impose an ordering over the set of interpretations is that it gives us a way to define the minimality of a model when there are more than two truth values involved.

Definition 6.7 *Let P be a ground program, and $\langle \mathcal{T}, \leq\rangle$ an ordered set of truth values. Then, the order $\langle \mathfrak{I}(P, \mathcal{T}), \leq\rangle$ is defined as follows:*

$$I_1 \leq I_2 \iff \forall A \in At(P) : I_1(A) \leq I_2(A) \ . \tag{43}$$

If $I_1 \leq I_2$ and $I_1 \neq I_2$, then we write $I_1 < I_2$.

Here we immediately note that $\langle \mathfrak{I}(P, \mathcal{T}), \leq \rangle$ is a poset since the relation $\leq$ is reflexive, transitive, and antisymmetric. Reflexivity and transitivity are trivial, and we see the antisymmetry by noting that the relation $\langle \mathcal{T}, \leq \rangle$ is antisymmetric.

Definition 6.8 *Given a ground program P, a valuator $\mathfrak{v}$, and an ordered set of truth values $\langle \mathcal{T}, \leq \rangle$ then a $\mathfrak{v}$-model M of P is* minimal *if there does not exist a model $M' \in \mathcal{A}_{\mathfrak{v}}(P)$ such that $M' < M$.*

The second concept encountered in the definition of the stable semantics is the justification property: each atom true in a model should have some reason be in there; if we cannot show that an atom A has to be true in a model, then it should be false. When we take this intuition and extend it to cover the case where there are possibly more than two truth values, it translates into finding models that are minimal in the sense of Definition 6.8.

On page 6 of Section 2 we defined the T_P operator that operates over sets of atoms and whose fixpoint corresponds to the minimal model of a set of monotonic Horn rules. On page 22 we then saw a similar operator for Horn constraint rules. Now, we generalize the notion and say that a class of rules has the *Horn property* if a set P of such rules always has a unique minimal model and the model can be found as the least fixed point of an operator Φ_P.

When we are seeking for a suitable fixpoint operator we are assisted by the fact that we can construct a complete lattice out of the order $\langle \mathfrak{I}(P, \mathcal{T}), \leq \rangle$ by giving suitable definitions for the binary *join* $(x \vee y)$ and *meet* $(x \wedge y)$ operators. By the well-known Knaster-Tarski fixpoint theorem, each order-preserving operator on a complete lattice has a least fixed point[6] so when given such an operator we only have to prove that the fixpoint corresponds to the minimal model.

Definition 6.9 *Let $I_1, I_2 \in \mathfrak{I}(P, \mathcal{T})$. Then, the operators $\vee$ and $\wedge$ are defined as follows:*

$$\begin{aligned} I_1 \vee I_2 &= \{(A, t) \mid A \in At(P) \text{ and } t = \max(I_1(A), I_2(A))\} \\ I_1 \wedge I_2 &= \{(A, t) \mid A \in At(P) \text{ and } t = \min(I_1(A), I_2(A))\} \ . \end{aligned} \tag{44}$$

Example 6.4 *In case of classical truth values $\mathcal{T}_B$ where an interpretation I can be identified with the set of atoms $M = \{a \mid I(a) = T\}$ that it makes true, the operations $I_1 \vee I_2$ and $I_1 \wedge I_2$ can be identified with the union $M_1 \cup M_2$ and the intersection $M_1 \cap M_2$.*

Proposition 6.1 *Let P be a ground program, and $\langle \mathcal{T}, \leq \rangle$ be a lattice of truth values. Then, the structure $\langle \mathfrak{I}(P, \mathcal{T}), \leq; \vee, \wedge \rangle$ is a complete lattice.*

Proof. In case of $\langle \mathfrak{I}(P, \mathcal{T}), \leq; \vee, \wedge \rangle$ we see that both $I_1 \vee I_2$ and $I_1 \wedge I_2$ assign a truth value for each atom in $At(P)$ and thus they are interpretations in $\mathfrak{I}(P, \mathcal{T})$, so $\langle \mathfrak{I}(P, \mathcal{T}) \rangle$ is a lattice.

Next, given a set of interpretations $S \subseteq \mathfrak{I}(P, \mathcal{T})$, we define two interpretations, $I_{\min}$ and $I_{\max}$ as follows:

$$\begin{aligned} \forall a \in At(P) : I_{\min}(a) &= \min\{I(a) \mid I \in S\} \cup \{\bot\} \\ \forall a \in At(P) : I_{\max}(a) &= \max\{I(a) \mid I \in S\} \cup \{\top\} \ . \end{aligned}$$

We see that $I_{\min} = \bigwedge S$ and $I_{\max} = \bigvee S$, so the lattice is complete. □

[6] See, for example, discussion in Chapter 7 in [14].

Definition 6.10 *Let P be a ground program and $\mathfrak{v}$ a valuator. Then, an operator $\Phi_P : \mathfrak{I}(P, \mathcal{T}) \rightarrow \mathfrak{I}(P, \mathcal{T})$ is* order-preserving *if and only if:*

$$\forall I_1, I_2 \in \mathfrak{I}(P, \mathcal{T}) : I_1 \leq I_2 \Rightarrow \Phi_P(I_1) \leq \Phi_P(I_2) \ . \tag{45}$$

Definition 6.11 *Given a valuator $\mathfrak{v}$, a class of ground rules $H \subseteq Rules(\mathcal{P}, \mathfrak{C})$ has the* Horn property under $\mathfrak{v}$ *if and only if the following two conditions hold:*

1. *P has a unique minimal $\mathfrak{v}$-model for each set of rules $P \subseteq H$; and*
2. *there exists an order-preserving operator Φ such that for all sets $P \subseteq H$, the least fixed point of Φ_P equals to the minimal $\mathfrak{v}$-model of P.*

The minimal model of a set $P \subseteq H$ is denoted by $cl_{\mathfrak{v}}(P)$.

6.4 Stable Models

We define the stable model semantics for general constraint literal programs via the notion of a reducer. A reducer is a function that creates a Gelfond-Lifschitz-style reduct out of a $Rules(\mathcal{P}, \mathfrak{C})$ program with respect to a given interpretation. The rules that belong to the reduct should have the Horn property as defined in the last section so that we can compute their minimal model using an order-preserving operator. If this minimal model agrees with the given interpretation and it satisfies all rules of the original program, then it is stable. For the rest of this section, we use the notation $Rules_H(\mathcal{P}, \mathfrak{C})$ to denote the class of rules that can belong to the reduct.

Definition 6.12 *Given a ground program P, an ordered set of truth values $\mathcal{T}$, and a signature $\langle \mathcal{P}, \mathfrak{C} \rangle$, a* constraint reducer $\mathfrak{c}$ *is a function:*

$$\mathfrak{c} : Lits(\mathcal{P}, \mathfrak{C}) \times \mathfrak{I}(P, \mathcal{T}) \rightarrow 2^{Lits(\mathcal{P}, \mathfrak{C})} \tag{46}$$

that associates a set of constraint literals $\mathfrak{c}(C, I)$ with each pair $\langle C, I \rangle$ where C is a constraint literal and I is an interpretation.

A reducer $\mathfrak{r}$ *is a function*

$$\mathfrak{r} : Rules(\mathcal{P}, \mathfrak{C}) \times \mathfrak{I}(P, \mathcal{T}) \rightarrow 2^{Rules_H(\mathcal{P}, \mathfrak{C})} \tag{47}$$

that associates a set of rules $\mathfrak{r}(r, I)$ for each pair $\langle r, I \rangle$ where r is a rule and I is an interpretation.

Example 6.5 *Given an extended cardinality constraint*

$$C = \langle \text{card}, \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}, \langle L, U \rangle \rangle$$

and a set of basic literals M, we can define the reducer $\mathfrak{c}$ for it as:

$$\mathfrak{c}(C, M) = \{\langle \text{card}, \{a_1, \ldots, a_n\}, \langle L - |\{b_i \mid b_i \notin M|, \infty \rangle \rangle\} \ . \tag{48}$$

This definition corresponds to the one given in Definition 4.14. Similarly, we can construct the reducer $\mathfrak{r}$ for a rule $r = C_0 \leftarrow C_1, \ldots, C_n$ as:

$$\begin{aligned} \mathfrak{r}(r, M) = \{p \leftarrow \mathfrak{c}(C_1, M), \ldots, \mathfrak{c}(C_n, M) \mid p \in C_0 \cap M \\ \wedge M \models_{\mathfrak{v}_C} C_1, \ldots, C_n\} \end{aligned} \tag{49}$$

where $\mathfrak{v}_C$ is defined as in Example 6.3.

After a reducer has been defined, we can generalize the concept of stable models for instantiated constraint programs.

Definition 6.13 *Given an ordered set of truth values $\mathcal{T}$, a valuator $\mathfrak{v}$, a satisfier $\mathfrak{s}_\mathfrak{v}$, a reducer $\mathfrak{r}$, and a ground program P, an interpretation $I \in \mathfrak{I}(P, \mathcal{T})$ is a* $\mathfrak{v}, \mathfrak{r}$-stable model *if and only if:*

1. $\forall r \in P : I \models_\mathfrak{v} r$; *and*
2. $cl_\mathfrak{v}(\bigcup_{r \in P} \mathfrak{r}(r, I)) = I$.

In the rest of this section we show how we can use valuators and reducers to extend the language of cardinality constraint programs.

6.5 Variables in Cardinality Constraint Bounds

We can start extending the cardinality constraints by allowing the use of variables and interpreted functions that evaluate to numbers in the rule bounds. Thus, now we allow upper and lower bounds of the form $f(X, Y)$ as long as it evaluates to an integer that can then be used as a cardinality bound. We will also take the approach that if either of the bounds does not evaluate to a valid integer, the cardinality constraint is not satisfied. In a practical programming system we would generate an error when this happens.

Definition 6.14 *Let $C = \langle \text{card}, \{S_1, \ldots, S_n\}, \langle L, U \rangle \rangle$ be a cardinality constraint where L and U are (possibly non-ground) terms, σ be a total substitution, D a data model, and M a set of ground atoms. Then, the valuator $\mathfrak{v}_{C'}(C, M, D, \sigma)$ is defined as follows:*

$$\mathfrak{v}_{C'}(C, M, D, \sigma) = \begin{cases} T, & \text{if } L\sigma, U\sigma \in \mathbb{N} \text{ and} \\ & L\sigma \leq |\{l \mid l \in E(C, D) \wedge M \models l\}| \leq U\sigma \\ F, & \text{otherwise} \end{cases} \quad (50)$$

where the expansion $E(C, D) = \bigcup_{i=1}^{n} E(S_i)$.

The constraint reducer may be defined otherwise as the one presented in (48), but it evaluates the lower bound L before performing the subtraction. As we always end up with normal expanded cardinality constraint literals, we do not have to define a new reducer or inference operator T.

Example 6.6 *Suppose that we have a program that plans routes for pipelines. One of the constraints of such a program would be that the route should not be too long. This constraint could be implemented as:*

$$\leftarrow M + 1 \leq \{\textit{used-pipe-segment}(P) : \textit{segment}(P)\}, \textit{max-length}(M)$$

If we want to make some further deductions on the number of used pipe segments, we can count their number by using the rule:

$$\textit{num-used}(N) \leftarrow N \leq \{\textit{used-pipe-segment}(P) : \textit{segment}(P)\} \leq N, d(N)$$

where $d(i)$ is defined to be true for $0 \leq i \leq n$ when n is the number of facts segment(p) that are true.

6.6 Weight Constraint Literals

Another straightforward way to extend cardinality constraints is to allow the literals to have an integral weight. Satisfaction is then defined in terms of the weights of the satisfied basic literals. A weight constraint W is of the form [65]:

$$L \leq [S_1 = w_1, \ldots, S_n = w_n] \leq U \tag{51}$$

where w_i are positive integral *weights*. In the notation of Definition 6.2 we would express (51) as:

$$W = \langle \text{weight}, \langle S_1, \ldots, S_n \rangle, \langle L, U, w_1, \ldots, w_n \rangle \rangle$$

The idea is now that the sum of weights of satisfied basic literals in the expansions of S_i has to be between L and U for the weight constraint to be satisfied. Each basic literal in the expansion of S_i gets w_i as its weight. We can define a valuator $\mathfrak{v}_W$ for an expanded weight constraint $W = L \leq [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \text{not } b_1 = w_{b_1}, \ldots, \text{not } b_m = w_{b_m}] \leq U$ as follows:

$$\mathfrak{v}_W(W, M) = \begin{cases} T, & \text{if } L \leq \sum_{a_i \in M} w_{a_i} + \sum_{b_i \notin M} w_{b_i} \leq U \\ F, & \text{otherwise} \end{cases} \tag{52}$$

The notion of a reduct for a weight constraint literal is defined analogously to cardinality constraint literals.

$$\mathfrak{c}_W(W, M) = L' \leq [a_1 = w_{a_1}, \ldots, a_n = w_{a_n}] \tag{53}$$

where the lower bound

$$L' = L - \sum_{b_i \notin M} w_{b_i} \quad .$$

The resulting rules are essentially identical with Horn constraint rules, so we can again use the same inference operator T_P.

Example 6.7 *Consider the weight constraint literal* $W = 2 \leq [a = 1, b = 2, \text{not } c = 3] \leq 3$. *Then, its reduct with respect to* $M_1 = \{b, c\}$ *is:*

$$2 \leq [a = 1, b = 2]$$

and the reduct with respect to $M_2 = \{a\}$ *is:*

$$0 \leq [a = 1, b = 2]$$

since $w_c = 2$ *and* $c \notin M_2$.

The weight constraint literals may be further extended by allowing the use of negative weights. Suppose that there is a literal $a = -w \in W$. Now, since having a true reduces the sum of the satisfied weights by w, we get the same result if we had not $a = w \in W$ and the bounds were $L + w$ and $U + w$. Thus, we can interpret negative weights as shortcuts for complementary literals and we do not have to worry about them. Note that this is only one of

the many possible semantics for negative weights. However, a discussion on the different possibilities is out of the scope of this work.

In case of non-ground programs it is very useful to allow variables in weight constraints. They may be defined using a similar mechanism that was used in the previous section to add variables to cardinality constraint bounds. In this case we also have to evaluate the weights during instantiation.

Example 6.8 *Consider the problem of programming a robot to do grocery shopping. Then, the following rule would state that the sum of the costs of purchased items should not exceed the money reserve:*

$$\leftarrow R + 1\ [\mathit{buy}(\mathit{What}, \mathit{Cost}) : \mathit{item}(\mathit{What}, \mathit{Cost}) = \mathit{Cost}], \mathit{reserve}(R)$$

As long as the second argument of item/2 is always a number, the atom weights are well-defined.

6.7 Classical Negation

When stable models are defined in the usual way, the negations are interpreted as negation-as-failure. That is, $\mathrm{not}(A)$ is true if we cannot prove that A is true. However, in some cases it is desirable to have a stronger notion of negation. Thus, we now extend our language to allow the use of *classical negation* $\neg A$ in literals.

The literal $\neg A$ differs from $\mathrm{not}(A)$ in that $\neg A$ is true only if we can prove that A is false. As a practical case where the difference matters, consider the case where we want to cross a road. We want to be certain that a car is not coming so we want to prove $\neg \mathit{car}$ before crossing. Here having $\mathrm{not}(\mathit{car})$ is not enough since there may be many reasons why *car* would not be true even if a car was coming. For example, we might decide to close our eyes so that we cannot see the approaching car and get $\mathrm{not}(\mathit{car})$ true that way.

In [25] Gelfond and Lifschitz presented a way to add classical negation for logic programs under the stable model semantics. Our definition is based on this approach but there is a small difference: we do not allow inconsistent stable models as they are difficult to implement properly and they do not correspond to valid solutions of problems. A set of basic literals M is *inconsistent* if for some atom A, both $A \in M$ and $\neg A \in M$.

We now extend our definitions of basic literals by stating that if A is an atom, then A, $\neg A$, not A, and not $\neg A$ are basic literals. Of these, we say that A and $\neg A$ are *positivistic* since they do not contain default negation. With this definition we can define the valuator and the reducer for constraint literals exactly as in the case of normal cardinality constraint programs. However, we have to alter the definition of the rule reducer slightly as we want to forbid inconsistent stable models.

Definition 6.15 *Let M be a set of positicistic basic literals and $r = C_0 \leftarrow C_1, \ldots, C_n$ a ground cardinality constraint rule. Then the reducer $\mathfrak{r}_\neg(r, M)$ is defined as follows:*

$$\mathfrak{r}_\neg(r, M) = \begin{cases} \mathfrak{r}(r, M), & \text{if } M \text{ is consistent,} \\ \emptyset, & \text{if } M \text{ is inconsistent .} \end{cases} \tag{54}$$

Example 6.9 *Let P be the program:*

$$\begin{aligned} 1 \leq \{\neg a\} &\leftarrow 1 \leq \{\text{not } b\} \\ 1 \leq \{a\} &\leftarrow 1 \leq \{\text{not } b\} \\ 1 \leq \{b\} &\leftarrow 1 \leq \{\text{not } \neg a\} \end{aligned}$$

Then the unique stable model is $M_1 = \{b\}$. The other candidate $M_2 = \{a, \neg a\}$ is not consistent, so $\mathfrak{r}(P, M_2) = \emptyset$ whose minimal model is empty, so M_2 is not stable.

6.8 Partial Models

In this section we consider how a more complicated semantics, namely the three-valued partial stable model semantics [53, 54], can be constructed for cardinality constraint programs using valuators and reducers. Earlier an atom had to be either true or false in a model, but now we allow that it may have an undefined value.

We start by defining our set of truth values. We will use the set

$$\mathcal{T}_p = \{F, U, T\}$$

where U represents the undefined truth value. The truth values are ordered so that

$$F < U < T \ .$$

A *partial* interpretation assigns one of the three truth values for each atom. A partial interpretation I divides a set of basic literals S into three subsets:

$$\begin{aligned} I^+(S) &= \{l \in S \mid I(l) = T\} \\ I^-(S) &= \{l \in S \mid I(l) = F\} \\ I^u(S) &= \{l \in S \mid I(l) = U\} \end{aligned}$$

We represent a partial interpretation I of P as the set $\{a \mid a \in I^+(At(P))\} \cup \{\text{not } a \mid\in I^-(At(P))\}$.

In [54] the reduct is defined with the help of a special atom $\mathbf{u}$ whose truth value is undefined in all interpretations. The atom $\mathbf{u}$ is used in reducts to signify negative literals $\text{not}(A)$ whose truth values are undefined. Here we follow this example but the set nature of cardinality constraints demands that we actually use a countably infinite set $\{\mathbf{u}_i \mid i \in \mathbb{N}\}$ of undefined atoms.

Next, we define the valuator for cardinality constraints and rules. A cardinality constraint is true if the number of true literals in it meets the lower bound and the total number of true and undefined literals does not exceed the upper bound. It is false if the number of true and undefined literals is less than the lower bound or the number of true literals is greater than the upper bound. If neither of these conditions hold, then it is undefined. A rule is true if the truth value of its head is greater or equal than the truth value of its body, and false otherwise. So, a rule may not have undefined value.

Definition 6.16 *Let $C = L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \leq U$ be a ground cardinality constraint, I be a partial interpretation. Then, the* valua-

tor $\mathfrak{v}_p(C, I)$ *is defined as follows:*

$$\mathfrak{v}_p(C, I) = \begin{cases} T, & \text{if } L \leq |I^+(C)| \text{ and } |I^+(C)| + |I^u(C)| \leq U \\ F, & \text{if } |I^+(C)| + |I^u(C)| < L \text{ or } |I^+(C)| > U \\ U, & \text{otherwise} \end{cases} \tag{55}$$

Let $r = C_0 \leftarrow C_1, \ldots, C_n$ *be a rule, then* $\mathfrak{v}_p(r, I)$ *is defined as follows:*

$$\mathfrak{v}_p(r, I) = \begin{cases} T, & \text{if } \mathfrak{v}_p(C_0, I) \geq \mathfrak{v}_p(C_i, I) \text{ for all } 1 \leq i \leq n \\ F, & \text{otherwise } . \end{cases} \tag{56}$$

If $\mathfrak{v}_p(r, M) = T$ *for all* $r \in P$*, then* $M \models_p P$*.*

Example 6.10 *Consider the ground rule* r_1*:*

$$r_1 = 1 \leq \{a, b\} \leftarrow 1 \leq \{c, \text{not } d, e\} \leq 2$$

where $C_0 = 1 \leq \{a, b\}$ *and* $C_1 = 1 \leq \{c, \text{not } d, e\} \leq 2$*. Let us define a partial interpretation* $I = \{a, c, \text{not } d\}$*. Now,*

$$\begin{array}{ll} |I^+(C_0)| = |\{a\}| = 1 & |I^+(C_1)| = |\{\text{not } d, c\}| = 2 \\ |I^-(C_0)| = |\emptyset| = 0 & |I^-(C_1)| = |\emptyset| = 0 \\ |I^u(C_0)| = |\{b\}| = 1 & |I^u(C_1)| = |\{e\}| = 1 \end{array}$$

As $|I^+(C_0)| > 1$ *and* $|I^+(C_0)| + |I^u(C_0)| < \infty$*,* $\mathfrak{v}_p(C_0, I) = T$*. On the other hand,* $|I^+(C_1)| + |I^u(C_1)| = 3 > 2$*, so* $\mathfrak{v}_p(C_1, I)$ *is not true. Since* $|I^+(C_1)| < 2$*, it is not false either, so* $\mathfrak{v}_p(C_1, I) = U$*. Now the rule* r_1 *is satisfied by* I *(*$I \models_p r_1$*) since* $\mathfrak{v}_p(C_0, I) \geq \mathfrak{v}_p(C_1, I)$*.*

Consider further the rule r_2*:*

$$r_2 = 1 \leq \{c, a, d\} \leq 1 \leftarrow \{b, d\} \leq 1$$

Now $\mathfrak{v}_p(C_0, I) = F$ *and* $\mathfrak{v}(C_1, I) = U$*. Thus,* r_2 *is not satisfied (*$I \not\models_p r_2$*).*

A partial interpretation M is a *partial model* of a program P if M satisfies all rules in it. Furthermore, M is a *minimal partial model* if it holds that there does not exist a partial model M' such that $M' < M$.

Next we want to define the reducers for constraint literals and rules. Intuitively, a constraint reducer works in two phases for $C = L \leq \{l_1, \ldots, l_n\} \leq U$:

1. The upper bound is removed by replacing C by two constraint literals:

$$C' = L \leq \{l_1, \ldots, l_n\}$$
$$C'' = n - U \leq \{\overline{l_1}, \ldots, \overline{l_n}\} \ .$$

   The lower bound of C'' is true exactly when the upper bound of C is true.

2. Each negative literal is removed from both C' and C'' as usual. However, each negative literal not b_i whose truth value is undefined is replaced by the atom $\mathbf{u}_i$.

Definition 6.17 *Let $S = \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\}$ be a set of literals and I a partial interpretation. Then, the functions lower and upper are defined as follows:*

$$\begin{aligned} \mathit{lower}(S, I) &= \{a_1, \ldots, a_n\} \cup \{\mathbf{u}_i \mid b_i \in I^u(S)\} \\ \mathit{upper}(S, I) &= \{b_1, \ldots, b_m\} \cup \{\mathbf{u}_i \mid a_i \in I^u(S)\} \end{aligned} \tag{57}$$

Definition 6.18 *Given a constraint literal $C = L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \leq U$, and a partial interpretation I, reduct $\mathfrak{c}_p(C, M)$ is defined as follows:*

$$\mathfrak{c}_p(C, I) = \{L' \leq \mathit{lower}(C, I), U' \leq \mathit{upper}(C, I)\} \tag{58}$$

where

$$\begin{aligned} L' &= L - |\{\text{not } b_1, \ldots, \text{not } b_m\} \cap I^-(C)| \\ U' &= m + n - U - |\{a_1, \ldots, a_n\} \cap I^-(C)| \ . \end{aligned} \tag{59}$$

Example 6.11 *Let $C = 1 \leq \{a, b, \text{not } f, \text{not } d\} \leq 2$ and $I = \{a, \text{not } d\}$. Then,*

$$\begin{aligned} \mathit{lower}(C, I) &= \{a, b, \mathbf{u}_f\} \\ \mathit{upper}(C, I) &= \{f, d, \mathbf{u}_b\} \end{aligned}$$

and

$$\begin{aligned} L' &= 1 - |\{\text{not } d\}| = 0 \\ U' &= 2 + 2 - 2 - |\{a\}| = 1 \end{aligned}$$

so

$$\mathfrak{c}_p(C, I) = \{0 \leq \{a, b, \mathbf{u}_f\}, 1 \leq \{f, d, \mathbf{u}_b\}\} \ .$$

Definition 6.19 *Given a rule $C_0 \leftarrow C_1, \ldots, C_n$ and a partial interpretation I, the reduct $\mathfrak{r}_p(r, I)$ is defined as follows:*

$$\mathfrak{r}_p(r, I) = \{a \leftarrow \mathfrak{c}_p(C_1, I), \ldots, \mathfrak{c}_p(C_n, I) \mid a \in C_0 \wedge I(a) \geq U\} \ . \tag{60}$$

The reduct $\mathfrak{r}_p(P, I)$ of a program P is the set $\{\mathfrak{r}_p(r, I) \mid r \in P\}$.

Example 6.12 *Let $I = \{a, \text{not } b\}$. Consider the rule r:*

$$1 \leq \{a, e\} \leq 2 \leftarrow 2 \leq \{\text{not } b, c, \text{not } d\} \leq 3 \ .$$

Now, $\mathfrak{r}_p(r, I)$ contains the following two rules:

$$\begin{aligned} a &\leftarrow 1 \leq \{c, \mathbf{u}_d\}, 1 \leq \{b, \mathbf{u}_c, d\} \\ e &\leftarrow 1 \leq \{c, \mathbf{u}_d\}, 1 \leq \{b, \mathbf{u}_c, d\} \end{aligned}$$

The rules of a reduct are Horn constraint rules, and each set of such rules has a unique minimal partial model.

Theorem 6.1 *Let P be a set of Horn constraint rules. Then, P has a unique minimal model.*

Proof. This proof is analogous to one presented in [54]. We denote by P_{pos} the program that is obtained by replacing all undefined atoms $\mathbf{u}_i$ by the truth value T and by P_{neg} the one obtained by replacing them by the truth value F.

Let M_{pos} be a minimal total model of P_{pos}. We see that M_{pos} is also a model of P_{neg} since the rules of P_{neg} that have satisfiable bodies all belong to P_{pos}. Let M_{neg} be any minimal total model of P_{neg} such that $M_{\text{neg}} \leq M_{\text{pos}}$. Now, the interpretation:

$$M = M^+_{\text{neg}}(At(P)) \cup \{\text{not } a \mid a \in M^-_{\text{pos}}(At(P))\}$$

is a minimal partial model of P. To see that this is the case consider a rule r. First, suppose that all literals in the body of r are true in M. As M_{neg} is a model of P_{neg}, this implies that $r \in P_{\text{neg}}$ and that the head atom of r is also true in M so r is satisfied. Next, suppose that none of the body literals of r are false in M. This implies that all its body literals are true in M_{pos} so its head is true in M_{pos} so it is not false in M. Thus, M is a model of P.

Consider a partial model $M' \leq M$. Then, $M'^+(At(P)) \subseteq M^+_{\text{neg}}(At(P))$ and $M^-_{\text{pos}}(At(P)) \subseteq M'^-(At(P))$. As the only difference between programs P and $P_\neg$ is that all undefined atoms have been removed from constraint literals, no constraint literal may have a higher truth value in M' than in M_{neg}. Thus, $M^+_{\text{neg}}(At(P)) \subseteq M'^+(At(P))$. We can do a similar analysis to compare M_{pos} to M'^- and see that $M'^-(At(P)) \subseteq M^-_{\text{pos}}(At(P))$ so $M' = M$ and the minimal model is unique. □

Theorem 6.2 *Let P be a set of Horn constraint rules. Then, the unique minimal partial model $M = \bigwedge \mathcal{A}_p(P)$.*

Proof. First we show that M is a partial model. Suppose that this is not the case. Then, there exists a rule $r : C_0 \leftarrow C_1, \ldots, C_n$ such that $M(C_0) < \min(\{M(C_i) \mid i \in [1,n]\})$. As for all $I \in \mathcal{A}_p$, $I \models_p r$, we know that $I(C_0) \geq \min(\{I(C_i) \mid i \in [1,n]\})$. However, as $(\bigwedge \mathcal{A}_p(P))(C_i) = \min(\{I(C_i) \mid I \in \mathcal{A}_p(P)\})$, it necessarily holds that $M(C_0) \geq \min(\{M(C_i) \mid i \in [1,n]\})$, a contradiction so M is a model.

Next, suppose that there exists a model $M' < M$. Then, there exists an atom a such that $M'(a) < M(a)$. Now $(M' \wedge M)(a) = M'(a) < M(a) = \bigwedge \mathcal{A}_p(P)(a)$, another contradiction. □

We now define an order-preserving operator $\widehat{T}$ for the three valued logic.

Definition 6.20 *Let P be a set of Horn constraint rules. Then the operator $\widehat{T}_P : \mathfrak{I}(P, \mathcal{T}_p) \to \mathfrak{I}(P, \mathcal{T}_p)$ is defined as follows:*

$$\begin{aligned}\widehat{T}_P(I) = \{\, a \mid \exists a \leftarrow C_1, \ldots, C_n \in P : I(C_i) = T \text{ for all } i \in [1,n]\} \cup \\ \{\, \text{not } a \mid \nexists a \leftarrow C_1, \ldots, C_n \in P : I(C_i) \geq U \text{ for all } i \in [1,n]\} \ .\end{aligned}$$

Theorem 6.3 *The operator $\widehat{T}_P$ is order-preserving.*

Proof. Suppose that $\widehat{T}_P$ is not order-preserving. Then there are two interpretations I_1 and I_2 such that $I_1 \leq I_2$ and $\widehat{T}_P(I_1) > \widehat{T}_P(I_2)$. This implies that there exists an atom a such that $I_1(a) \leq I_2(a)$ and $\widehat{T}_P(I_1)(a) > \widehat{T}_P(I_2)(a)$.

However, for all rules $a \leftarrow C_1, \ldots, C_n$ it holds that $I_2(C_i) \geq I_1(C_i)$, so also $\widehat{T}_P(I_2)(a) \geq \widehat{T}_P(I_1)(a)$ as all rules are Horn constraint rules, which causes a contradiction. Thus, $\widehat{T}_P$ is order-preserving. □

Corollary 6.1 *The operator $\widehat{T}_P$ has a least fixed point.*

Example 6.13 *Let P be the program:*

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow 1 \leq \{a, f, e\} \\ f &\leftarrow 1 \leq \{\boldsymbol{u}_f\} \ . \end{aligned}$$

Now,

$$\begin{aligned} \widehat{T}_P(\bot) &= \{a, \text{not } e\} \\ \widehat{T}_P(\{a, \text{not } e\}) &= \{a, b, \text{not } e\} \\ \widehat{T}_P(\{a, b, \text{not } e\}) &= \{a, b, \text{not } e\} \end{aligned}$$

and a fixpoint is reached.

Proposition 6.2 *Let P be a set of Horn constraint rules and I a partial interpretation. Then, $\widehat{T}_P(I)(a) = \max(\{\min(\{I(C_1), \ldots, I(C_n)\} \cup \{T\}) \mid a \leftarrow C_1, \ldots C_n \in P\} \cup \{F\})$.*

Proof. Follows directly from Definition 6.20. □

Proposition 6.3 *Let P be a set of Horn constraint rules. Then I is a partial model of P if and only if $\widehat{T}_P(I) \leq I$.*

Proof. Suppose that $I \not\models_p P$. Then there exists a rule $r = C_0 \leftarrow C_1, \ldots, C_n$ such that $I(C_0) < \min(\{I(C_i) \mid i \in [1, n]\})$. Thus

$$\widehat{T}_P(I)(C_0) \geq \min(\{I(C_i) \mid i \in [1, n]\}) > I(C_0)$$

and $\widehat{T}_P(I) > I$. Similarly, if $\widehat{T}_P(I) > I$, then there exists a rule that is not satisfied in I so I is not a model. □

Theorem 6.4 *Given a set of Horn constraint rules P, the least fixed point of the operator $\widehat{T}_P$ is the unique minimal partial model $M = \bigwedge \mathcal{A}_p(P)$ of P.*

Proof. By Proposition 6.3 $M' = \text{lfp}(\widehat{T}_P(\bot)) \in \mathcal{A}_p(P)$. So, we know that $M \leq M'$ and we have only to show that $M' \leq M$. Suppose that this is not the case, that is, $M' > M$. Then there exists an atom a such that $M'(a) > M(a)$ and for all rules $a \leftarrow C_1, \ldots, C_n$, $M'(a) > \min(\{M(C_i) \mid i \in [1, n]\})$. However, this is impossible since by Proposition 6.2 $\widehat{T}_P(I)(a) = \max(\{\min(\{I(C_1), \ldots, I(C_n)\} \cup \{T\}) \mid a \leftarrow C_1, \ldots C_n \in P\} \cup \{F\})$. Thus, $M' \leq M$. □

Now we are finally ready to define partial stable models of cardinality constraint programs.

Definition 6.21 *A partial interpretation M is a* partial stable model *of a ground cardinality constraint program P iff*

1. *$M \models_p P$; and*
2. *M is the least fixed point of $\widehat{T}_{(\mathfrak{r}_p(P,M))}$.*

Example 6.14 *Consider the following ground program P:*

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow 1 \leq \{a, \text{not } f\} \leq 1 \\ f &\leftarrow 1 \leq \{\text{not } f\} \end{aligned}$$

We immediately see that this program has no stable models, since the third rule causes a contradiction. However, it has a unique partial stable model $M_1 = \{a\}$. We notice that it satisfies all rules since it satisfies the head of the first rule and in the last two rules all constraint literals are undefined. Its reduct $\mathfrak{r}_p(P, M_1)$ is:

$$\begin{aligned} a &\leftarrow \\ b &\leftarrow 1 \leq \{a, \mathbf{u}_f\}, 1 \leq \{\mathbf{u}_f\} \\ b &\leftarrow 1 \leq \{f\} \\ f &\leftarrow 1 \leq \{\mathbf{u}_f\} \end{aligned}$$

We see that $cl(\mathfrak{r}_p(P, M_1)) = \{a\} = M_1$.

As the last thing in the section we note that partial models of cardinality constraint programs are a true generalization of partial models as defined in [54]. There, the reduct r^I of a normal rule r with respect to a partial interpretation I is obtained by replacing each negative literal not a by the truth value $\overline{I(a)}$ where $\overline{T} = F$, $\overline{F} = T$ and $\overline{U} = U$.

Theorem 6.5 *Let P be a normal logic program. Then, I is a minimal partial model of P^I if and only if it is a minimal partial model of $\mathfrak{c}_p(P, I)$.*

Proof. Consider a rule $r = h \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m$. When the negative literals are interpreted as a shortcuts of cardinality constraint literals $1 \leq \{\text{not } b_i\}$, we see that their reducts $\mathfrak{c}_p(b_i, I)$ are

$$\mathfrak{c}_p(b_i, I) = \begin{cases} 1 \leq \{\}, & I(b_i) = T \\ 1 \leq \{\mathbf{u}_{b_i}\}, & I(b_i) = U \\ 0 \leq \{\}, & I(b_i) = F \end{cases}$$

We see that these reducts are equivalent to the reducts in the case of r^I, so the reducts P^I and $\mathfrak{c}_p(P, I)$ are equivalent and have the same minimal partial model. □

7 TRANSLATIONS FOR SEMANTIC EXTENSIONS

A number of different semantics for logic programs has been proposed during recent years. The basic aim of these new semantics is to provide a framework for solving some problem that is difficult to solve using the existing ones. However, without a working implementation of the semantics it is of little practical use. Also, it is possible that in some cases the semantics gives unintended results and these cases may be difficult to find without an implementation. Often the fastest way to construct a prototype implementation for a semantics is to define a translation from it to some previously implemented semantics.

Here we will use a notation where $\mathcal{C}$ denotes a class of logic programs and $\mathcal{A}_{\mathcal{C}}(P)$ denotes the set of all models of a program $P \in \mathcal{C}$ under the semantics of $\mathcal{C}$. A translation τ is a function that transforms a logic program $P \in \mathcal{C}$ into a program $\tau(P) \in \mathcal{C}'$ [29]. We are interested in translations that are faithful in the sense that the models of the translation $\tau(P)$ have to correspond to the models of the original program P and that given a model of $\tau(P)$ we can identify which atoms are true in the corresponding model of P.

Definition 7.1 *Translation $\tau : \mathcal{C} \to \mathcal{C}'$ is* faithful *if and only if for all programs $P \in \mathcal{C}$:*

1. *there exists an injective function $f : At(P) \to At(\tau(P))$; and*

2. *there exists a bijection $g : \mathcal{A}_{\mathcal{C}}(P) \to \mathcal{A}_{\mathcal{C}'}(\tau(P))$ such that for all models $M \in \mathcal{A}_{\mathcal{C}}(P)$ and basic literals l,*

$$l \in M \Leftrightarrow f(l) \in g(M) \ .$$

Even though the faithfulness is the most important property of a translation [29], there are two other conditions that make translations more usable in practice:

- *Modularity.* A translation τ is *modular* if $\tau(A \cup B) = \tau(A) \cup \tau(B)$ for all logic programs $A, B \in \mathcal{C}$.

- *Polynomiality.* A translation τ is *polynomial* if it can be computed in a polynomial time with respect to the length of P.

In this section we examine how we can translate various extensions of the basic semantics into normal cardinality constraint programs. Namely, we give translations for the classical negation semantics. In addition we show how cardinality constraint programs can be used to implement normal logic programs with ordered disjunction [7, 10] using a two-program translation. Again, we give the translations only for ground programs.

Even though it would be possible to give a similar translation also for weight constraint programs we do not give it since it is much more efficient to alter the stable model computation engine to support weights directly.

7.1 Classical Negation

Cardinality constraint programs with classical negation can be implemented along the lines presented in [25]. For each atom a we add a new predicate atom a' to denote the classical negation $\neg a$. The basic idea then is to keep the program otherwise as it was but to add a number of rules of the form:

$$\leftarrow a, a'$$

to weed out inconsistent stable models. In practice,

Definition 7.2 *Let l be an extended basic literal. Then, the function $f_\neg(l)$ is defined as follows:*

$$f_\neg(l) = \begin{cases} a(t_1,\ldots,t_n), & \text{if } l = a(t_1,\ldots,t_n) \\ a'(t_1,\ldots,t_n), & \text{if } l = \neg a(t_1,\ldots,t_n) \\ \text{not } a(t_1,\ldots,t_n), & \text{if } l = \text{not } a(t_1,\ldots,t_n) \\ \text{not } a'(t_1,\ldots,t_n), & \text{if } l = \text{not } \neg a(t_1,\ldots,t_n) \end{cases}$$

Let $C = L \leq \{l_1,\ldots,l_n\} \leq U$ be an extended cardinality constraint literal. Then, the translation $f_\neg(C)$ is the standard cardinality constraint literal:

$$f_\neg(C) = L \leq \{f_\neg(l_1),\ldots,f_\neg(l_n)\} \leq U \quad .$$

Definition 7.3 *Let P be a ground cardinality constraint program. Then, the translation $\tau_\neg(P)$ is the program:*

$$\begin{aligned} \tau(P) = {} & \{f_\neg(C_0) \leftarrow f_\neg(C_1),\ldots,f_\neg(C_n)\} \mid C_0 \leftarrow C_1,\ldots,C_n \in P\} \\ & \cup \{\leftarrow a', a \mid a \in At(P)\} \end{aligned} \tag{61}$$

Theorem 7.1 *The translation $\tau_\neg$ is faithful, modular and polynomial for all ground programs.*

Proof.

- *Faithfulness.* First we note that $f_\neg$ is an injective function from $At(P)$ to $At(\tau(P))$ so the first condition is satisfied. Next, suppose that M is a stable model of P. By Definition 6.15 M has to be consistent since if it is not, then by definition $cl(\mathfrak{c}_\neg(P,M)) = \emptyset \neq M$. Thus, for all atoms a, either $a \notin M$ or $\neg a \notin M$ so every rule $\leftarrow a', a$ is satisfied in $\tau(P)$. As other rules of $\tau(P)$ are equivalent to rules in P, the set $\{f_\neg(l) \mid l \in M\}$ is a stable model of $\tau(P)$. The same argument works also in the other direction.

- *Modularity.* It is easy to see that the translation is modular.

- *Polynomial.* We can create $\tau(P)$ by going through P once changing all negative literals $\neg a$ to a' and adding the rules $\leftarrow a, a'$ for them. This takes a linear time.

□

7.2 Preferences

In some cases not all stable models are the same. For example, if the stable models of a program correspond to valid plans on how to travel from a place A to B, then we might prefer to a fast route to a slower one, or perhaps a cheap route to an expensive one.

When we formalize such preference information, we get a preference semantics for logic programs. A large body of literature on preferences has been published during the last decade, with new different semantics being defined every year.

The great number of different priority semantics makes it difficult to present a comprehensive overview on the subject. One such overview appears in [8] but it does not contain any recent work.

The different preference semantics can be divided into two classes depending on whether the preferences are assigned over the rules [6, 9, 15, 57, 27] or over the atoms [59, 7].

In this section we take one of the many proposed, namely Brewka's logic programs with ordered disjunction [7] (denoted $LPOD$) and show how they can be implemented using ω-restricted programs. This presentation extends the one given in [10]. Note that the semantics is presented here in a slightly different way than in [7] and [10].

An ordered disjunction is a rule of the form:

$$C_1 \times \cdots \times C_n \leftarrow A_1, \ldots, A_m, \text{not } B_1, \ldots, \text{not } B_k \tag{62}$$

where the C_i are all atoms and A_i, B_i are all atoms. The intuition here is that if the body is true, then at least one of the atoms C_i has to be true. The atoms C_i are listed in the order of preference, so if it is possible to have C_1 true, it should be chosen. If not, then C_2 and so on until if all other choices are not possible, then at least C_n has to be true.

Given a set P of ground rules of the form (62) and a set of ground atoms M, then the reduct P^M is the set:

$$\begin{aligned} P^M = \{\ & C_i \leftarrow A_i, \ldots, A_m \mid \\ & C_1 \times \cdots \times C_n \leftarrow A_1, \ldots, A_m, \text{not } B_1, \ldots, B_k \in P, \\ & C_i \in M \wedge M \cap \{C_1, \ldots, C_{i-1}, B_i, \ldots, B_k\} = \emptyset\} \end{aligned} \tag{63}$$

A set M of atoms is a stable model of P if M satisfies all rules of the program and it is the least model of P^M.

Example 7.1 *Let P be the LPOD:*

$$\begin{aligned} r_1 &: a \times b \leftarrow \text{not } c \\ r_2 &: b \times c \leftarrow \text{not } d \end{aligned}$$

Then, P has three stable models: $M_1 = \{a, b\}$, $M_2 = \{c\}$, $M_3 = \{b\}$. Their reducts are as follows:

$$\begin{aligned} P^{M_1} &= \{a \leftarrow; b \leftarrow\} \\ P^{M_2} &= \{c \leftarrow;\} \\ P^{M_3} &= \{b \leftarrow\} \end{aligned}$$

On the other hand, $M_4 = \{b, c\}$ is not a stable model since its reduct is:

$$P^{M_4} = \{b \leftarrow\} \ .$$

Next, we define a preference relation over the set $\mathcal{A}(P)$ of all models of P. Three different preference criteria are defined in [10] and here we consider only the simplest one, Pareto-optimality. The intuition is that we prefer a stable model M_1 of P over M_2 if M_1 satisfies at least one rule of P better than M_2 and it satisfies all the rest rules of P at least as well as M_2.

We start formalizing this criterion by defining how well a model M satisfies a rule r.

Definition 7.4 *Let M be a stable model of an LPOD P. Then M satisfies the rule*

$$C_1 \times \cdots \times C_n \leftarrow A_1, \ldots, A_m, \text{not } B_1, \ldots, \text{not } B_k$$

- *to degree 1 if $A_j \notin M$, for some j, or $B_i \in M$, for some i; or*
- *to degree j $(1 \leq j \leq n)$ if all $A_j \in M$, no $B_i \in M$, and $j = \min\{r \mid C_r \in M\}$.*

The degrees can be seen as penalties: the lower the degree, the better satisfied we are. If the body of a rule is not satisfied, then the choice does not come into play and we can assign the best degree 1 to it. We denote the degree of r in M by $deg_M(r)$.

Example 7.2 *Let r_1 be the rule*

$$r_1 : a \times b \leftarrow \text{not } c$$

from Example 7.1. Furthermore, let $M_1 = \{a, b\}$, $M_2 = \{c\}$, and $M_3 = \{b\}$. Then,

$$\begin{aligned} deg_{M_1}(r) &= 1 \\ deg_{M_2}(r) &= 1 \\ deg_{M_1}(r) &= 2 \ . \end{aligned}$$

Next, we define the actual Pareto criterion as follows.

Definition 7.5 *Let M_1 and M_2 be stable models of an LPOD P. Then M_1 is* Pareto-preferred *to M_2 ($M_1 >_p M_2$) iff there is $r \in P$ such that $deg_{M_1}(r) < deg_{M_2}(r)$, and for no $r' \in P$ $deg_{M_1}(r') > deg_{M_2}(r')$.*

Example 7.3 *Let P be as in Example 7.1, $M_1 = \{a, b\}$, $M_2 = \{c\}$, and $M_3 = \{b\}$. Then, $M_1 >_p M_3$ as $degs_{M_1}(r_1) = 1 < 2 = degs_{M_3}(r_2)$ while $degs_{M_1}(r_2) = 1 = degs_{M_3}(r_2)$. Similarly, $M_1 >_p M_2$. However, models M_2 and M_3 are incomparable since they satisfy different rules to the first degree.*

Definition 7.6 *A set of literals M is a Pareto-preferred stable model of an LPOD P iff M is a stable model of P and there is no stable model M' of P such that $M' >_p M$.*

From this on we will drop out the Pareto qualifier whenever we discuss preferred stable models.

Example 7.4 *In Example 7.1 the only preferred stable model is* $M_1 = \{a, b\}$.

Now we would like to have a translation from $LPOD$s to cardinality constraint programs. Since the existence of a preferred stable model for a ground $LPOD$ is $\mathbf{\Sigma_2^P}$-complete [10] while the same problem for cardinality constraint programs is **NP**-complete [65], a polynomial translation from $LPOD$s to cardinality constraints can exists only if the polynomial hierarchy collapses. It would be possible to define an exponential translation between them, but that would make computing preferred models even more intractable that it currently is.

Even though a single polynomial time translation is probably impossible, we can translate an $LPOD$ P into two cardinality constraint programs:

- A *generator* $G(P)$ whose stable models correspond to the stable models of P; and
- A *tester* $T(P, M)$ for checking whether a given stable model M of P is preferred.

The two programs are run in an interleaved fashion. First, the generator constructs a stable model M of P. Next, the tester tries to find a stable model set M' that is strictly better than M. If there is no such M', we know that M is a preferred stable model. Otherwise, we use $G(P)$ to construct the next candidate. When we want to find only one preferred stable model we can save some effort by taking M' directly as the new candidate.

The basic idea of $G(P)$ is to add a number of bookkeeping atoms to P to ensure that each rule r is used to derive only at most one atom to the model. In practice, this is done by adding a number of atoms of the form $c(r, k)$ to denote the case where we choose to use the rule r to derive the atom C_k.

To make model comparison easier we also add another set of new atoms, $s(r, k)$ to denote that the rule r is satisfied to the degree k. These atoms are not strictly necessary but they make the programs more readable.

Definition 7.7 *Let P be an $LPOD$ and $r = C_1 \times \cdots \times C_n \leftarrow body$ be a rule in P. Then the translation $G(r, k)$ of the kth option of r is defined as follows:*

$$G(r,k) = \{C_k \leftarrow c(r,k), \text{not } C_1, \ldots, \text{not } C_{k-1}, body; \tag{64}$$
$$\leftarrow \text{not } c(r,k), C_k, \text{not } C_1, \ldots, \text{not } C_{k-1}, body\} \tag{65}$$

The satisfaction translation $S(r)$ is:

$$S(r) = \{s(r,1) \leftarrow \text{not } c(r,1), \ldots, \text{not } c(r,n)\} \cup \tag{66}$$
$$\{\, s(r,i) \leftarrow c(r,i) \mid 1 \leq i \leq n\} \tag{67}$$

The translation $G(r)$ is:

$$G(r) = \big\{1 \leq \{c(r,1), \ldots, c(r,n)\} \leq 1 \leftarrow body\big\} \cup \bigcup\{G(r,k) \mid 1 \leq k \leq n\} \cup S(r) \tag{68}$$

The generator $G(P)$ is defined as follows:

$$G(P) = \bigcup\{G(r) \mid r \in P\} \tag{69}$$

Also, the reason for having two rules in $G(r,k)$ may not be clear, since (64) already ensures that only correct stable models of the program P' will be generated. To see why it is necessary, consider the situation where some C_j, $j < k$, is a consequence of a different part of the program. Then without (65) we could have a stable model where $c(r,k)$ is true, but C_k is not because C_j blocks (64). In other words, we would have chosen to satisfy r to the degree k, but it actually would be satisfied to the degree j. The rule (65) prevents this unintuitive behavior by always forcing us to choose the lowest possible degree.

Example 7.5 *The program P in Example 7.1 is translated to:*

$$
\begin{array}{ll}
1\,\{c(1,1), c(1,2)\}\,1 \leftarrow \text{not } c & a \leftarrow c(1,1), \text{not } c \\
1\,\{c(2,1), c(2,2)\}\,1 \leftarrow \text{not } d & b \leftarrow c(2,1), \text{not } d \\
\leftarrow \text{not } c(1,1), a, \text{not } c & b \leftarrow c(1,2), \text{not } a, \text{not } c \\
\leftarrow \text{not } c(1,2), b, \text{not } a, \text{not } c & c \leftarrow c(2,2), \text{not } b, \text{not } d \\
\leftarrow \text{not } c(2,1), b, \text{not } d & s(1,1) \leftarrow \text{not } c(1,1), \text{not } c(1,2) \\
\leftarrow \text{not } c(2,2), c, \text{not } b, \text{not } d & s(2,1) \leftarrow \text{not } c(2,1), \text{not } c(2,2) \\
s(1,1) \leftarrow c(1,1) \quad s(1,2) \leftarrow c(1,2) & s(2,1) \leftarrow c(2,1) \quad s(2,2) \leftarrow c(2,2)
\end{array}
$$

The three stable models are:

$$
\begin{aligned}
M_1 &= \{a, b, c(1,1), c(2,1), s(1,1), s(2,1)\} \\
M_2 &= \{b, c(1,2), c(2,1), s(1,2), s(2,1)\} \\
M_3 &= \{c, c(2,2), s(1,1), s(2,2)\}\ .
\end{aligned}
$$

Note that in the last case there is no atom $c(1,k)$ since the body of the first rule is not satisfied.

Proposition 7.1 *Let P be an LPOD. If M is a stable model of $G(P)$, then $M \cap At(P)$ is a stable model of P. If M is a stable model of P, then there exists a set of atoms $A \subseteq At(G(P)) \setminus At(P)$ such that $M \cup A$ is a stable model of $G(P)$.*

Proof. Let M be a stable model of P. Now, for each rule $r = C_1 \times \cdots \times C_n \leftarrow$ *body* define $p(M,r) = \{c(r,k), s(r,k) \mid$ *body* is satisfied in $M, C_k \in M,$ and $\forall i < k : C_i \notin M\} \cup \{s(r,1) \mid r \in P$ and *body* is unsatisfied in $M\}$. Let $M' = M \cup \bigcup_{r \in P} p(M,r)$. By definition of $p(M,r)$, M' satisfies all rules (64)–(67). Finally, (68) is satisfied since exactly one atom $c(r,k)$ was added to M' for each rule r that had its body true. Also, each atom $c(r,k)$ and $s(r,k)$ that occur in M' occur in a head of a rule whose body is satisfied in the reduct of $G(P)$ so they belong to the deductive closure and M' is stable.

Now, suppose that M' is a stable model of $G(P)$. Then for each atom $C_k \in M = M' \cap Lit(P)$, there exists a rule r' of the form (64) where C_k is the head and some atom $c(r,k) \in M'$ occurs positively in the body and $c(r,i) \notin M'$ for all $i < k$. The rule r' belongs to the translation of some rule $r \in P$. When we take the reduct of the rule r with respect to M we get the rule:

$$r^M = C_k \leftarrow A_1, \ldots, A_n$$

since no atom C_i, $i < k$ may be in M. As the only way to derive the atom $c(r,k)$ to M' is that the body of r is true, we see that $C_k \in cl(P^M)$. Thus, for all atoms $C \in M$, $C \in cl(P^M)$ so M is a stable model of P. □

Next we consider the tester program $T(P,M)$. Since we want the tester to find a stable model M' that is strictly better than a given M, we define two new atoms, namely *better* and *worse*.[7] The intuition is as follows: *better* (*worse*) is true when M' is in some aspect better (worse) than M. If both are true, then the models are incomparable.

Definition 7.8 *Let P be an LPOD. Then the tester $C(T,M)$ is defined as follows:*

$$\begin{aligned} T(P,M) = G(P) &\cup \{o(r,k) \mid s(r,k) \in M\} \cup \{\mathit{rule}(r) \leftarrow \mid r \in P\} \\ &\cup \{\mathit{degree}(d) \leftarrow \mid \exists r \in P \text{such that } r \text{ has at least } d \text{ options}\} \\ &\cup \{\leftarrow \text{not } \mathit{better};\ \leftarrow \mathit{worse}, \\ &\quad \mathit{better} \leftarrow s(R,I), o(R,J), I < J, rule(R), \\ &\qquad\qquad degree(I), degree(J) \\ &\quad \mathit{worse} \leftarrow s(R,J), o(R,I), I < J, rule(R), \\ &\qquad\qquad degree(I), degree(J)\} \end{aligned}$$

Proposition 7.2 *Let P be an LPOD and M be a stable model of $G(P)$. If M' is a stable model of $T(P,M)$, then $M' \cap \mathit{At}(P)$ is a stable model of P that is preferred to M. If there exists a stable model M' of P that is preferred to M then there exists a set of atoms A such that $M' \cup A$ is stable model of $T(P,M)$.*

Proof. First, suppose that $T(P,M)$ has a stable model M'. Then $\mathit{better} \in M'$ and $\mathit{worse} \notin M'$. We see that *better* is true exactly when $\exists r : deg_{M'}(r) < deg_M(r)$. Since $\mathit{worse} \notin M'$, we know that $\neg\exists r : deg_{M'}(r) > deg_M(r)$ so $M' \cap \mathit{At}(P) >_p M$. Conversely, if there exists $M' >_p M$, then M' is generated by the $G(P)$ part of $T(P,M)$ so $M' \cup \bigcup\{p(M',r) \mid r \in P\} \cup \{\mathit{better}\}$ is a stable model of $T_p(P,M)$. □

The following corollary is immediate from Proposition 7.2:

Corollary 7.1 *Let P be an LPOD and M be a stable model of $G(P)$. Then M is preferred if and only if $T(P,M)$ does not have any stable models.*

[7] This translation could also be made without these two atoms but they make the program more readable.

8 COMPUTATIONAL COMPLEXITY

In this section we examine the computational complexity of ω-restricted programs. This section expands [75] where most of these results were originally proven. We are mainly interested in two different problems:

- INSTANTIATION: Given an ω-restricted program P, a data model D_0, and a ground atom $p(t_1, \ldots, t_n)$ (where p is a domain predicate), does $D_P \models p(t_1, \ldots, t_n)$ hold?

- MODEL: Given an ω-restricted program P and a data model D_0, does $\langle P, D_0 \rangle$ have a stable model?

Intuitively, INSTANTIATION tells us how difficult it is to create the relevant instantiation of a program with variables and MODEL then tells how hard it is to find a stable model of a program with variables.

As a data model D_0 may be infinite, we represent it by a Turing machine that computes the interpretations of function symbols and data predicates. The intuition is that whenever we have to compute a value of a function, we give the function symbol and its arguments as input to the Turing machine and let it compute the value. Similarily, if we give a ground atom as an input, the machine gives the truth value of the atom as its output.

Throughout this section we use as our data model the Herbrand interpretation $D_{P,H}$ of the program P as defined in Section 4.1. We do this because our definition of a data model allows us to use arbitrary complex functions in programs and their evaluation may dominate the instantiation process. Thus, we limit us to the simplest possible case where function evaluation does not significantly affect the necessary time to solve INSTANTIATION or MODEL.

In particular, the initial data model $D_{P,H}$ has the following properties:

1. There are no pure data predicates. Instead, all predicate symbols are program predicates and domain predicates are used to give values to all variables.

2. All function symbols have the trivial interpretation $I(f(a)) = f(a)$.

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:

- The maximum number of variables is fixed to some constant d or it is unlimited; and

- The programs can contain arbitrary function symbols or only 0-ary constants.

If the number of variables is fixed, then each rule in a program may contain at most d global variables, and each literal set in the rule body may contain at most d local variables.

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

Table 2: Computational complexity

P	Polynomial time (n^k)
NP	Non-deterministic polynomial time
EXP	Exponential time (2^{n^k})
NEXP	Non-deterministic exponential time
2-**EXP**	Doubly exponential time ($2^{2^{n^k}}$)
2-**NEXP**	Non-deterministic doubly exponential time

Figure 9: Complexity Classes

The main complexity results[8] are presented in Table 2. The MODEL complexity for ground weight constraint programs with the stable model semantics has been presented in [65] and for normal logic programs in [45, 13]. Since ω-restricted programs are essentially a subclass of more general weight constraint rules, the complexity results of [65] apply. The meanings of the complexity classes are shown in Figure 9. For a comprehensive account on computational complexity see, for example, [52].

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 $p(t_1, \ldots, t_n)$ we can create a program P' such that P' has a stable model only if $D_P \models p(t_1, \ldots, t_n)$. The program P' is defined as follows:

$$P' = P \setminus P_\omega \cup \{\leftarrow \text{not } p(t_1, \ldots, t_n)\} \ .$$

In effect, we remove all rules that belong to the ω-stratum, and check whether the query atom $p(t_1, \ldots, t_n)$ is a logical consequence

Theorem 8.1 *Problems* INSTANTIATION *and* MODEL *are decidable for finite ω-restricted programs.*

Proof. First note that for a ground ω-restricted program INSTANTIATION amounts to evaluating all terms in it, computing the minimal model of domain definitions $\bigcup_{i \in \mathbb{N}} P_i$, and then checking whether a given atom a is satisfied in the minimal model, and this is decidable since by definition the interpretation I has to be computable.

Also, if we have a finite ground ω-restricted program, MODEL is decidable since there are only a finite number of possible model candidates and we can check them all.

[8]The article [75] contains an error in the fixed-variable non-constant function symbols allowed case. There it was claimed that they were **NEXP**-complete while the correct result is that they are 2-**NEXP**-complete.

Next, we show by induction that given a finite ω-restricted program P and a data model D, the sizes of relevent ground instantiations of strata program P_i, $i \in \mathbb{N}$, are all finite, and so their stable models M_i can be computed. Since only a finite number of P_i are nonempty, this implies that the union $M = \bigcup_{i \in \mathbb{N}} M_i$ is also finite. It is enough to consider the relevant instantiation since by Theorem 5.2 both $\mathbf{HI}(P_\omega, D_P)$ and $\mathbf{HI}_r(P_\omega, D_P)$ have the same stable models.

Consider first the stratum program P_0. Since all rules on the 0-stratum have to be ground, we see that $\mathbf{HI}_r(P_0, D)$ is finite. Hence, M_0 is finite, so in $D_0 = D \uplus M_0$ the interpretations of domain predicates are finite.

Next, suppose that $\mathbf{HI}_r(P_i, D_{i-1})$ is finite when $1 \leq i \leq n$ for some $n \in \mathbb{N}$.

Now, if P_{n+1} is empty, then its instantiation is also empty and $M_{i+1} = \emptyset$. Otherwise, consider a rule $r \in P_{n+1}$. Each variable that occurs in r has to occur also on a domain literal l that belongs to some stratum $i \leq n$. By induction hypothesis the interpretation of $pred(l)$ is finite, so $\mathbf{HI}_r(r, D_n)$ is also finite.

We can generate this instantiation by systematically going through the interpretations of the domain literals in the rule body and creating all possible substitutions that correspond to the interpretations, and instantiate the rule using them. As P_{n+1} is finite, also $\mathbf{HI}_r(P_{n+1}, D_n)$ has to be finite.

A naive algorithm that computes the relevant instantiation of $\langle P, D_0 \rangle$ is presented in Figure 10. □

Note that this result holds for an arbitrary data model D since by definition a data model may not contain uncomputable functions.

We can now see that the computational complexity of a subclass of ω-restricted programs is directly related to the number of relevant rules in their instantiations.

8.1 Turing Machine Translation

We establish the complexity hardness results in this work by proving that the computations of a Turing machine M can be simulated by a logic program $P(M)$. The basic idea is that the configurations of M are encoded as sets of atoms of $P(M)$ so that $P(M)$ has a stable model where the atoms corresponding to an accepting configuration exactly when that configuration is reachable from the initial configuration of M.

The program consists of two parts, a fixed one that implements the transition function of an arbitrary Turing machine 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 size of the input-specific part is polynomial in the size of x.

Definition 8.1 *A deterministic* Turing machine *is a quadruple*

$$M = (K, \Sigma, \delta, s)$$

where K is a finite set of states, Σ is a finite alphabet containing the blank symbol $\sqcup$, $s \in K$ is the initial state and δ is a transition function $\delta : K \times \Sigma \to (K \cup \{y, n\}) \times \Sigma \times \{-1, 0, 1\}$.

function *instantiate*(P, D_0)
    $\mathcal{S} :=$ *create_stratification*(P)
    $D := D_0$
    for $i := 0$ **to** number of non-omega strata in $\mathcal{S}$ **do**
        $P' :=$ *instantiate_rules*($P_i^{\mathcal{S}}$, D)
        $M := cl(P')$
        $D := D \uplus M$
    end for
    $P' :=$ *instantiate_rules*($P_\omega^{\mathcal{S}}$, D)
    return $\langle P', D \rangle$
end function

function *instantiate_rules*(Ruleset R, Data D)
    $S := \emptyset$
    foreach $r \in R$ **do**
        $S' := \{E(r', D) \mid r' \in inst(r, D)$ and $D \models body_s(r')\}$
        $S := S \cup S'$
    end foreach
    return S
end function

Figure 10: Simple instantiation algorithm

A computation of a Turing machine M given an input x starts from the configuration $(s, \sqcup x)$ and each computation step yields a new configuration according to δ until one of the halting states y (*accept*) or n (*reject*) is reached.

We encode the states of a Turing machine M using the predicate *state*(q), the alphabet using *symbol*(σ), and the transition function using the predicate *transition*($q_1, \sigma_1, q_2, \sigma_2, d$), where $d \in \{l, s, r\}$ denoting left, stationary, and right, respectively. The atom *at-place*(σ, p, t) is used to denote that the tape cell p contains the symbol σ at the time step t. The predicate *current-state*(q, p, σ, t) indicates that the machine is in the state q and the head is over the tape cell p looking at the symbol σ at the time step t.

We encode one computation step using the two rules:

$$\begin{aligned} \textit{at-place}(S_2, P, T') \leftarrow\ & \textit{transition}(Q_1, S_1, Q_2, S_2, D), \\ & \textit{current-state}(Q_1, P, S_1, T), \\ & \textit{place}(P), \textit{time}(T), \textit{time}(T'), \\ & \textit{next}(T, T') \end{aligned} \tag{70}$$

$$\begin{aligned} \textit{current-state}(Q_2, P', S_3, T') \leftarrow\ & \textit{transition}(Q_1, S_1, Q_2, S_2, D), \\ & \textit{next-place}(P, D, P'), \\ & \textit{current-state}(Q_1, P, S_1, T), \\ & \textit{at-place}(S_3, P', T), \textit{time}(T), \\ & \textit{time}(T'), \textit{place}(P), \textit{symbol}(S_3), \\ & \textit{next}(T, T')\ . \end{aligned} \tag{71}$$

The rules above handle the cell where the read/write-head is currently po-

sitioned. In addition, we have to assert that the state of the other tape cells stays constant:

$$\begin{aligned} \textit{at-place}(S_1, P_1, T') \leftarrow\ & \textit{current-state}(Q, P_2, S_2, T), \textit{next(T, T')}, \\ & \textit{at-place}(S_1, P_1, T), \textit{time}(T), \textit{time(T')}, \\ & \textit{symbol}(S_1), \textit{symbol}(S_2), \textit{state}(Q), \\ & \textit{place}(P_1), \textit{place}(P_2), \text{not } \textit{equal}(P_1, P_2) \ . \end{aligned} \tag{72}$$

In the initial configuration all tape cells that are not part of the input are empty:

$$\textit{at-place}(\sqcup, P, 1) \leftarrow \textit{place}(P), \text{not } \textit{part-of-input}(P) \ . \tag{73}$$

The first $|x|$ tape cells are initialized from the input and they also belong to the extension of *part-of-input*/1.

The predicate *next-place*/3 connects the adjacent tape cells together so that the read/write-head can be moved to the correct direction:

$$\begin{aligned} \textit{next-place}(P_1, \textit{right}, P_2) &\leftarrow \textit{next}(P_1, P_2) \\ \textit{next-place}(P_1, \textit{left}, P_2) &\leftarrow \textit{next}(P_2, P_1) \\ \textit{next-place}(P, \textit{stationary}, P) &\leftarrow \textit{place}(P) \ . \end{aligned} \tag{74}$$

Finally, we want to recognize whether the Turing machine halts in an accepting state or not:

$$\begin{aligned} \textit{accept} &\leftarrow \textit{current-state}(y, P, S, T), \textit{place}(P), \textit{symbol}(S), \textit{time}(T) \\ \textit{reject} &\leftarrow \textit{current-state}(n, P, S, T), \textit{place}(P), \textit{symbol}(S), \textit{time}(T) \ . \end{aligned} \tag{75}$$

Note that we have not yet given definitions for the predicates *time*/1, *place*/1, *next*/2 that encode the time steps, tape cells, and the successor relation. In the following complexity proofs we show how we can define them in a polynomial number of rules using tools that are available for the four different ω-restricted program classes.

We can generalize the translation to allow non-deterministic Turing machines by forcing the machine to choose between possible transitions at all computation steps. This can be done by changing the rules (70) and (71) to:

$$\begin{aligned} \textit{at-place}(S_2, P, T') \leftarrow\ & \textit{chosen}(Q_1, S_1, Q_2, S_2, D, T) \\ & \textit{transition}(Q_1, S_1, Q_2, S_2, D), \\ & \textit{current-state}(Q_1, P, S_1, T), \\ & \textit{place}(P), \textit{time}(T), \textit{time}(T'), \\ & \textit{next(T, T')} \end{aligned} \tag{70'}$$

$$\begin{aligned} \textit{current-state}(Q_2, P', S_3, T') \leftarrow\ & \textit{chosen}(Q_1, S_1, Q_2, S_2, D, T) \\ & \textit{transition}(Q_1, S_1, Q_2, S_2, D), \\ & \textit{next-place}(P, D, P'), \\ & \textit{current-state}(Q_1, P, S_1, T), \\ & \textit{at-place}(S_3, P', T), \textit{time}(T), \\ & \textit{time}(T'), \textit{place}(P), \textit{symbol}(S_3), \\ & \textit{next(T, T')} \ . \end{aligned} \tag{71'}$$

and adding the rule:

$$\begin{aligned} 1 \leq \{\rho Q_2 S_2 D.\langle \mathit{chosen}(Q_1, S_1, Q_2, S_2, D, T) : \\ \mathit{transition}(Q_1, S_1, Q_2, S_2, D)\rangle\} \leq 1 \leftarrow \\ \mathit{current\text{-}state}(Q_1, P, S_1, T), \\ \mathit{place}(P), \mathit{time}(T) \end{aligned} \tag{76}$$

A similar translation for normal logic programs has been presented earlier by V. W. Marek and J. B. Remmel in [40].

8.2 Complexity Results for Omega-Restricted Programs

In this section we give our main complexity results. All proofs are divided into two parts, *inclusion* and *hardness*. When proving inclusion, we show that the problem belongs to some complexity class. We establish it by showing that the size of the ground instantiation does not grow faster than some upper bound. When proving hardness, we show that we can translate an arbitrary deterministic Turing machine that solves some problem in the complexity class and its input into an ω-restricted program where $D_P \models \mathit{accept}$ exactly when the machine accepts its input. This translation is polynomial with respect to the size of the input of the machine.

Before presenting the results we have to define what the size of a program means. In the proofs we make a simplifying assumption that each literal and non-constant function symbol occurring in a program has the same number of arguments and that every rule has equally many literals in it.

Definition 8.2 *The* size *of a term t is defined inductively as follows:*

1. *for all 0-ary constants c, $\mathit{size}(c) = 1$; and*
2. *for all compound terms $f(t_1, \ldots, t_n)$, $n > 0$,*

$$\mathit{size}(f(t_1, \ldots, t_n)) = 1 + \sum_{i=1}^{n} \mathit{size}(t_i) \ .$$

Definition 8.3 *Let P be an ω-restricted program. Then, its* size *is defined to be:*

$$\mathit{size}(P) = m \cdot d \cdot \ell \cdot T \tag{77}$$

where:

- *m is the number of rules in P;*
- *d is the arity of predicates and non-constant function symbols of P;*
- *ℓ is the number of basic literals in each rule; and*
- *T is the maximum size of a term that occurs in P.*

We will also use the convention that the upper limit for variables occurring in a rule or a literal set is equal to the maximum arity d.

In some cases parts of the definition of $\mathit{size}(P)$ are constant and may therefore be left out since we are interested only in the asymptotic behavior.

When we instantiate a program P, the size explosion of $\mathbf{HI}_r(P)$ 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:

$$p(X_1, \ldots, X_n) \leftarrow d(X_1), \ldots, d(X_n) \ .$$

Instantiating a rule of this form results in t^n ground instances if there are t ground terms in the extension of $d/1$.

There are two more factors that increase the size of the instantiated program but whose contributions are small compared with the two cases above:

3. When a literal set $l(X_1, \ldots, X_n) : d(X_1), \ldots, d(X_n)$ is expanded, the resulting ground rule will have t^n ground instances of the literal l in its body. The reason why this does not affect as much as the first case is that the particular ground rule can be used to derive only one ground atom, so this size increase does not propagate to predicates that are derived using the head of the rule.

4. When a function symbol f is applied to the terms t_1, ..., t_n, then the size of the new term is the sum of the sizes of the argument terms.

Theorem 8.2 INSTANTIATION *of an ω-restricted program is* **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 $\mathbf{HI}_r(P_{i+1})$ for some 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. Similarity, the expansion of a literal set 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 it can be computed in polynomial time using the algorithm *instantiate*.

(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 [13]. Such programs are a special case of ω-restricted programs. Thus, it is **P**-hard to decide whether $D_P \models a$.

□

Theorem 8.3 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 a polynomial time and it has a polynomial number of rules so we can non-deterministically guess a stable model and verify it in a polynomial time.

(b) *Hardness.* MODEL for ground normal logic programs is **NP**-complete [45] and they are a special case of ω-restricted programs.

□

Theorem 8.4 *The* INSTANTIATION *problem 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 $\mathit{size}(P) = n$. Suppose that there are c different ground terms in P. We now prove by induction over $d = 1, 2, \ldots$ where d is the maximum number of variables in rules and literal sets that the maximum size of the instantiation of P is:

$$\mathit{size}(\mathbf{HI}_r(P)) \leq md\ell c^{2d} \leq dn^{2n+2}$$

where m is the number of rules and ℓ the number of literals in each rule and $\mathit{size}(P) = n = md\ell$. (Since no new ground terms are generated during the instantiation, the maximum term length T is constant and may be left out of consideration.) Thus, it turns out that the size of the instantiation grows $O(2^{n^2})$.

We divide our analysis in two parts. First, we show that the number of ground instances of rules in $\mathbf{HI}_r(P)$ is at most mc^d, and then we show that each such 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,

$$|\mathbf{HI}_r(r)| \leq c \ .$$

Next, suppose that for all programs that have at most k different variables $X_1, \ldots, X_k$ in them, $|\mathbf{HI}_r(r)| \leq c^k$ 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 $X_1, \ldots, X_k$ in the rules that were produced in the previous step.

The first step directly corresponds with 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:

$$|\mathbf{HI}_r(r)| \leq c^k \cdot c = c^{k+1}$$

so se 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 $\mathbf{HI}_r(P)$ is:

$$|\mathbf{HI}_r(P)| \leq mc^d \ .$$

We can make exactly the same induction to show that the expansion of a literal set occurring in the rule body may contain at most c^d basic literals, so the body of an instantiated rule may contain at most ℓc^d literals. Combining these two figures we get:

$$\mathit{size}(\mathbf{HI}_r(P)) \leq (mc^d) \cdot d \cdot (\ell c^d) = md\ell c^{2d} \leq dn^{2d+2} \ .$$

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 $\lg 2$ 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 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 a 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:

$$\mathit{number}(x_1, \ldots, x_{n^k}) \leftarrow \mathit{bit}(x_1), \ldots, \mathit{bit}(x_{n^k}) \ . \tag{78}$$

The predicate $\mathit{bit}/1$ is auxiliary with the extension $\{0, 1\}$. The successor relation can be encoded with the rule:

$$\begin{aligned} \mathit{next}(x_1, \ldots, x_{n^k}, y_1, \ldots, y_{n^k}) \leftarrow\ & \mathit{add}(x_1, 1, y_1, c_1), \\ & \mathit{add}(x_2, c_1, y_2, c_2), \\ & \quad \vdots \\ & \mathit{add}(x_{n^k}, c_{n^k-1}, y_{n^k}, c_{n^k}) \end{aligned} \tag{79}$$

where $add/4$ encodes one-bit addition and is defined using the following four facts:

$$\begin{array}{ll} add(0,1,1,0) \leftarrow & add(0,0,0,0) \leftarrow \\ add(1,0,1,0) \leftarrow & add(1,1,0,1) \leftarrow \quad . \end{array} \tag{80}$$

Now the time steps and tape positions can be defined in terms of numbers:

$$\begin{array}{l} time(x_1,\ldots,x_{n^k}) \leftarrow number(x_1,\ldots,x_{n^k}) \\ place(x_1,\ldots,x_{n^k}) \leftarrow number(x_1,\ldots,x_{n^k}) \ . \end{array} \tag{81}$$

Finally, we replace all references to $time/1$ and $place/1$ by $time/n^k$ and $place/n^k$ and add all necessary domain predicates to the rule bodies.

□

Theorem 8.5 *The* MODEL *problem of an unlimited-variable ω-restricted program is* **NEXP***-complete if no non-constant function symbols are allowed.*

Proof. As in Theorem 8.3. □

When we consider INSTANTIATION when we allow the use of 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 8.1 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 $d \geq 8$ and there are at least two different non-constant function symbols available.*

Proof. We need to construct a binary counter from 0 up to $2^{n^k} - 1$. We do this by encoding an m-bit binary number x as a function $b_1(b_2(\cdots b_m(0)\cdots))$ where b_i is f if the ith 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:

$$\begin{array}{l} number_m(t(X)) \leftarrow number_{m-1}(X) \\ number_m(f(X)) \leftarrow number_{m-1}(X) \end{array} \tag{82}$$

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:

$$number_0(0) \leftarrow \quad . \tag{83}$$

The successor relation can also be defined recursively:

$$\begin{array}{l} next_{i+1}(t(X),t(Y)) \leftarrow next_i(X,Y) \\ next_{i+1}(f(X),f(Y)) \leftarrow next_i(X,Y) \\ next_{i+1}(f(X),t(Y)) \leftarrow last_i(X), first_i(Y) \end{array} \tag{84}$$

where $last_i/1$ and $first_i/1$ are defined as:

$$\begin{aligned} last_i(t^i(0)) &\leftarrow \\ first_i(f^i(0)) &\leftarrow \quad . \end{aligned} \tag{85}$$

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. □

Theorem 8.6 INSTANTIATION *of a fixed-variable ω-restricted program that uses non-constant function symbols is 2-***EXP***-complete, if the maximum number of variables occuring in a rule or a literal set $d \geq 8$.*

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 8.2 *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 $\mathbf{HI}_r(P)$ is bounded above by $2^s d^s T$ where s is the number of 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 $size(t_0) = T \leq 2^0 d^0 T$. 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:

$$t_{k+1} = f(\underbrace{t_k, \ldots, t_k}_{d \text{ times}})$$

where f is a function symbol of maximal arity. We see that:

$$\begin{aligned} size(t_{k+1}) &= d \cdot size(t_k) + 1 \\ &\leq 2d \cdot size(t_k) \quad . \end{aligned}$$

Here we use the induction hypothesis to note that:

$$2d \cdot size(t_k) = 2d \cdot 2^k d^k T = 2^{k+1} d^{k+1} T$$

and the induction is complete. □

Lemma 8.3 *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 $i+1$ strata programs P_j, then for each rule $r \in P_{i+1}$ it holds that $|\mathbf{HI}_r(r)| \leq c^d$ 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. □

Lemma 8.4 *Let P be a cardinality constraint program with s strata with m rules each. Then, $\mathbf{HI}_r(P)$ 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 $i+1$ strata 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 8.3 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 exponentation we get:

$$\begin{aligned} 2m(2^{\sum_{i=0}^{k-1} d^i} \cdot m^{\sum_{j=0}^{k} d^k})^d &= 2m \cdot 2^{d\sum_{i=0}^{k-1} d^i} \cdot m^{d\sum_{j=0}^{k} d^j} \\ &= 2m \cdot 2^{\sum_{i=0}^{k-1} dd^i} \cdot m^{\sum_{j=0}^{k} dd^j} \\ &= 2m \cdot 2^{\sum_{i=0}^{k-1} d^{i+1}} \cdot m^{\sum_{j=0}^{k} d^{j+1}} \\ &= 2m \cdot 2^{\sum_{i=1}^{k} d^i} \cdot m^{\sum_{j=1}^{k+1} d^j} \\ &= 2^{1+\sum_{i=1}^{k} d^i} \cdot m^{1+\sum_{j=1}^{k+1} d^j} \\ &= 2^{\sum_{i=0}^{k} d^i} \cdot m^{\sum_{j=0}^{k+1} d^j} \end{aligned}$$

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 $\mathbf{HI}_r(P)$. □

Lemma 8.5 *The expansion of a literal set S 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 S.*

Proof. First we note that if S has d local variables, then there are at most c_{k-1}^d basic literals in the expansion of S where c_{k-1} is the number of ground atoms that can be derived using rules on the first k strata. By Lemma 8.4 we know that $c_k \leq 2^{d^k} \cdot m^{d^{k+1}}$. □

Proof. (of Theorem 8.6)

(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 $\mathit{size}(P) = n = sm\ell T$ 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 $\mathbf{HI}_r(P)$. By Lemmas 8.2–8.5 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:

$$\mathit{size}(\mathbf{HI}_r(P)) \leq 2^{d^s} m^{d^{s+1}} \cdot 2^{d^s} m^{d^{s+1}} \ell \cdot 2^s d^s T \ .$$

As each of s, m, ℓ, and T are less than n, we get that:

$$\mathit{size}(\mathbf{HI}_r(P)) \leq 2^{d^n} n^{d^{n+1}} \cdot 2^{d^n} n^{d^{n+1}} \ell \cdot 2^n d^n n \ .$$

When we rearrange the terms we see that:

$$\mathit{size}(\mathbf{HI}_r(P)) \leq 2^{2d^n+n} n^{2d^{n+1}+1} \leq n^{4d^{n+1}+1} \tag{86}$$

when $n \geq 2$. We now compare the growth rate of $f(n) = n^{4d^{n+1}+1}$ to the growth rate of $g(n) = 2^{2^{n^2}}$.

$$\begin{aligned} \lg f(n) &= (4d^{n+1} + 1) \lg n \leq 5d^{n+1} \lg n \\ \lg(5d^{n+1} \lg n) &= \lg 5 + (n+1) \lg d + \lg \lg n \\ \lg g(n) &= 2^{n^2} \\ \lg \lg g(n) &= n^2 \ . \end{aligned}$$

As n^2 grows strictly faster than $n + \lg \lg n$, we know that $\mathit{size}(\mathbf{HI}_r(P)) = O(2^{2^{n^2}})$, and 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 8.1

and implement the numbers as terms. However, in this case we start with n rules:

$$\begin{aligned} \mathit{number}_0(0) &\leftarrow \\ &\vdots \\ \mathit{number}_0(n) &\leftarrow \end{aligned}$$

Then, we create the further levels of numbers using:

$$\mathit{number}_{i+1}(f(X,Y)) \leftarrow \mathit{number}_i(X), \mathit{number}_i(Y) \ . \qquad (87)$$

The intuition is that number_1 is seen as a 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 $\mathit{number}_i/1$. 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:

$$\mathit{next}_0(i, i+1) \leftarrow$$

and continuing with a recursive definition:

$$\begin{aligned} \mathit{next}_{i+1}(f(X,Y), f(X,Z)) &\leftarrow \mathit{number}_i(X), \mathit{next}_i(Y,Z) \\ \mathit{next}_{i+1}(f(X,L), f(Y,F)) &\leftarrow \mathit{first}_i(F), \mathit{last}_i(L), \mathit{next}_i(X,Y) \ . \end{aligned}$$

Finally, we have:

$$\begin{aligned} \mathit{first}_0(0) &\leftarrow \\ \mathit{last}_0(n) &\leftarrow \\ \mathit{first}_{i+1}(f(X,X)) &\leftarrow \mathit{first}_i(X) \\ \mathit{last}_{i+1}(f(X,X)) &\leftarrow \mathit{last}_i(X) \end{aligned}$$

and the proof is complete.

□

We now present a simple example program that illustrates the doubly-exponential growth of ground instantiation.

Example 8.1 *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:

$$\begin{aligned} |\mathbf{HI}_r(P_0)| &= 2 = 2^{2^1-1} \\ |\mathbf{HI}_r(P_1)| &= 2 \cdot |\mathbf{HI}_r(P_0)|^2 = 2 \cdot 2^2 = 8 = 2^{2^2-1} \\ |\mathbf{HI}_r(P_2)| &= 2 \cdot |\mathbf{HI}_r(P_1)|^2 = 2 \cdot 8^2 = 128 = 2^{2^3-1} \\ |\mathbf{HI}_r(P_3)| &= 2 \cdot |\mathbf{HI}_r(P_2)|^2 = 2 \cdot 128^2 = 32768 = 2^{2^4-1} \ . \end{aligned}$$

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:

$$\begin{aligned} |\mathbf{HI}_r(P_0)| &= 3 = 3^{2^1-1} \\ |\mathbf{HI}_r(P_1)| &= 3 \cdot |\mathbf{HI}_r(P_0)|^2 = 3 \cdot 3^2 = 27 = 3^{2^2-1} \\ |\mathbf{HI}_r(P_2)| &= 3 \cdot |\mathbf{HI}_r(P_1)|^2 = 3 \cdot 27^2 = 2187 = 3^{2^3-1} \\ |\mathbf{HI}_r(P_3)| &= 3 \cdot |\mathbf{HI}_r(P_2)|^2 = 3 \cdot 2187^2 = 14,348,907 = 3^{2^4-1} \ . \end{aligned}$$

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 kth 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 8.7 *The* MODEL *problem for a fixed-variable ω-restricted program that uses function symbols is 2-***NEXP***-complete, if $d \geq 8$.*

Proof. As in Theorem 8.3. □

Theorem 8.8 *The* INSTANTIATION *problem of an ω-restricted program is 2-***EXP***-complete.*

Proof.

(a) *Inclusion.* In (86) in proof of Theorem 8.6 we saw that $size(\mathbf{HI}_r(P))$ is bounded above by:

$$size(\mathbf{HI}_r(P)) \leq n^{4d^{n+1}+1}$$

The same result holds also in this case, but here d is not fixed but is linear to n. Thus,

$$size(\mathbf{HI}_r(P)) \leq n^{4n^{n+1}+1}$$

and further:

$$\lg n^{4n^{n+1}+1} = (4n^{n+1}+1)\lg n \leq (5n^{n+1})\lg n$$
$$\lg(5n^{n+1}\lg n) = \lg 5 + (n+1)\lg n + \lg\lg n \ .$$

As $n\lg n + \lg\lg n = O(n^2)$, we see again that $size(\mathbf{HI}_r(P)) = O(2^{2^{n^2}})$ and the proof is complete.

(b) *Hardness.* Follows directly from Theorem 8.6.

□

Theorem 8.9 MODEL *for an ω-restricted program is 2-***NEXP***-complete.**

Proof. As in Theorem 8.3. □

$$
\begin{array}{rl}
h \leftarrow l_1, \ldots, l_n & \text{Basic Rule} \\
h \leftarrow L \leq \{l_1, \ldots, l_n\} & \text{Constraint Rule} \\
h \leftarrow L \leq [l_1 = w_1, \ldots, l_n = w_n] & \text{Weight Rule} \\
\{h_1, \ldots, h_n\} \leftarrow l_1, \ldots, l_n & \text{Choice Rule}
\end{array}
$$

Figure 11: Internal *smodels* rule types

9 IMPLEMENTATION

The ω-restricted subset of cardinality constraint programs has been implemented in the SMODELS [77] tool. SMODELS has two parts, *smodels* proper that implements the stable model semantics for ground programs and *lparse* that instantiates programs with variables. The tools are available for download at `http://www.tcs.hut.fi/Software/smodels`.

In addition to standard stable semantics SMODELS has also support for several other semantics including partial model [30] and preferred model [10] semantics. These semantics are implemented by translating the input programs into ω-restricted programs whose stable models coincide with the models under different semantics.

9.1 Smodels

The *smodels* started as an efficient logic engine for computing the stable model semantics for ground normal logic program and it has been augmented with three special rule types, *constraint*, *weight*, and *choice* rules. The exact combination of the new rule types were chosen so that the expressive power of constraint literals could be reached while still allowing fast truth-value propagation during computation.

A basic rule is the standard logic program rule of the form:

$$h \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m \tag{88}$$

where h, a_i, and b_i are all ground atoms. A constraint rule has the form

$$h \leftarrow L \leq \{a_1, \ldots, a_n, \text{not } b_1, \ldots, \text{not } b_m\} \tag{89}$$

where L is an integral lower bound and h, a_i, and b_i are again atoms. A weight rule is otherwise similar to a constraint rule but each basic literal in the body may have an integral weight attached to it:

$$h \leftarrow L \leq [a_1 = w_1, \ldots, a_n = w_n, \text{not } b_1 = w_{n+1}, \ldots, b_m = w_{n+m}] \tag{90}$$

Finally, a choice rule has the form:

$$\{h_1, \ldots, h_n\} \leftarrow a_1, \ldots, a_n, \text{not } b_1, \ldots, b_m \tag{91}$$

where h_i, a_i, and b_i are all atoms.

The implementation details have been presented in [65] and [64].

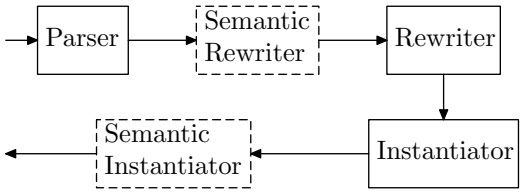


Figure 12: How *lparse* processes an input program

9.2 Lparse

The instantiator *lparse* has two main functions: it removes variables from user programs and it translates complex constructs into simpler ones. Conceptually *lparse* processes its input in three distinct phases:

1. *parser* reads the input programs and creates their dependency graphs;
2. *semantic rewriter* translates any rules with non-standard semantics into standard rules;
3. *rewriter* translates complex constructs into simple ones;
4. *instantiator* removes variables from rules; and
5. *semantic instantiator* instantiates control rules when a non-standard semantics is used.

In practice the different phases are interleaved so that, for example, in many simple cases rewriting happens on the fly in the instantiator.

Rewriter
Since the *smodels* engine understand only four internal rule types, we have to translate more complex constructs into these rules. In particular, an *smodels* rule may have at most one constraint literal in it. If it is in the head, there may be no bounds at all, and if it is in the body, it must be the only body literal and it may not have an upper bound. Thus, any rule that has a constraint literal with both bounds in it has to be translated to two *smodels* internal rules.

The basic idea of the translation is that if a constraint literal

$$L \leq \{l_1, \ldots, l_n\} \leq U$$

occurs in a rule body, it is replaced by two literals int_1 and not int_2 where neither int_1 nor int_2 occurs elsewhere in the program and two rules:

$$int_1 \leftarrow L \leq \{l_1, \ldots, l_n\}$$
$$int_2 \leftarrow U + 1 \leq \{l_1, \ldots, l_n\}$$

If there are any global variables in the constraint literal, then they are added as arguments to int_1 and int_2. The rewritten rules are not necessarily ω-restricted but we can get around that problem by using the domain predicates of the original rule to find out the relevant ground instances.

Example 9.1 *Consider the rule:*

$$a(X) \leftarrow 2 \leq \{\rho Y.\langle c(X,Y) : d(X,Y)\rangle\} \leq 3,$$
$$1 \leq \{\rho Z.\langle e(Z) : f(Z)\rangle\} \leq 2, f(X) \ .$$

This rule is translated into

$$\begin{aligned} a(X) &\leftarrow int_1(X), \text{not } int_2(X), int_3, \text{not } int_4, f(X) \\ int_1(X) &\leftarrow 2 \leq \{c(X,Y) : d(X,Y)\} \\ int_2(X) &\leftarrow 4 \leq \{c(X,Y) : d(X,Y)\} \\ int_3 &\leftarrow 1 \leq \{e(Z) : f(Z)\} \\ int_4 &\leftarrow 3 \leq \{e(Z) : f(Z)\} \end{aligned}$$

If the constraint literal occurs in the head, we drop the bounds from the original rule and add two new rules to ensure that the correct number of head atoms will be satisfied. So, a rule of the form

$$L \leq \{a_1, \ldots, a_n\} \leq U \leftarrow body$$

is translated to:

$$\begin{aligned} \{a_1, \ldots, a_n\} &\leftarrow body \\ &\leftarrow n - L + 1 \leq \{\text{not } a_1, \ldots, \text{not } a_n\}, body \\ &\leftarrow U + 1 \leq \{a_1, \ldots, a_n\}, body \end{aligned}$$

Here n is the number of ground basic literals in the expansions of literal sets in the head. Thus, this translation has to be done after they have been expanded. Also, if *body* is not empty, the two new rules have also to be rewritten.

Instantiator
The instantiator works in two phases:

1. domain generation; and

2. program instantiation

In the domain generation phase the domain predicates are identified automatically and their extensions computed. In the program instantiation phase the domain predicates are used as a data model to instantiate all program predicates.

The domain predicates are detected using the algorithm *create_stratification* that was presented in Figure 7 on page 34.

The rules are instantiated one at a time, starting from domain predicates on the lowest stratum and continuing up to the ω-stratum. In the case of domain predicates we then compute the deductive closure of their rules and store those instances in the data model. The rules for program predicates are output as they are created.

The same algorithm is used for both domain and program predicates. Conceptually, we create a natural join over the extensions of the domain

```
procedure instantiate(Rule R)
{ Create the lits array }
pos := 0
while pos ≥ 0 do
        instance := get_next_ground_instance(lits[pos])
        if bind_literal(lits[pos], instance) = T do
                if pos = max do
                        emit(R)
                else
                        pos := pos + 1
                endif
        else
                remove_binding(lits[pos])
                pos := pos − 1
        endif
endwhile
```

Figure 13: The *lparse* instantiator algorithm

predicates in the rule body, and then use that to find out all relevant variable bindings.

However, we do not create a relational table for the join explicitly. Instead, we find the possible bindings one row at a time and output a ground instance as soon as one row is completed. This has the advantage that it is quick when the rule bodies are relatively simple and it uses potentially less memory than computing the full tables. However, if many rules have almost identical bodies, we end up having to do unnecessary work as we have to create the same join again and again.

The rules are instantiated one at a time. When we instantiate a rule R we start by creating an array *lits* for storing the positive domain literals of the rule body. The domain literals are sorted in ascending order on the size of their extensions. Then we start an iteration where we find a ground instance of the first domain literal, bind the global variables according to it, and try to find a compatible instance of the next literal. This is continued until either all variables are bound or no compatible ground instance exists. In the latter case we backtrack and remove the latest variable bindings. This algorithm is shown in Figure 13. The variable *max* in the algorithm description is the index of the last domain literal of the rule.

Here we abstract away the details of functions *get_next_ground_instance* and *bind_literal*. The former iterates over the instantiation of the domain literals keeping an internal state, and returns a null-instance when all instances are processed. The latter binds the values of global variables to the instance in question and returns T if it succeeds and F if *instance* is a null-instance or contains conflicting values for already bound variables.

The actual implementation of the algorithm has some additional optimizations. For example, if there is a negative literal not A in the rule body, its truth value is checked as soon as all variables in it have been bound. If it turns out that A is necessarily true, we discard the partial instance. Also,

we create indices for the extensions where possible so that we do not have to always iterate through the whole extensions of all domain predicates.

Literals Sets
We also expand literal sets using the *instantiate* procedure. We take the atoms occurring in the condition to be the rule body and the set literal to be the head. The literal set is then replaced by the instantiated set of literals. A literal set is expanded as early as possible. If there are no global variables in it, it is expanded before the rule is instantiated. We can do this because those literal sets have always the same expansion.

10 EXAMPLES

In this section we give three examples of using ω-restricted cardinality constraint programs to model problems. The first problem is a classical planning puzzle where a number of people have to be transported over a river, and in the second one we want to find nondeterministic finite automata that satisfy certain structural requirements when they are determinised.

Even though both problem domains are rather simple, they are not toy problems. The optimal length of the solution for the planning puzzle is 16 steps long[9] and the search space is large enough that the most obvious encoding of the puzzle fails to find a solution. In the automata theoretic example we have to determinize a nondeterministic automaton as a subproblem, so the size of the instantiated program is guaranteed to be exponential with respect to size of the input.

We include the SMODELS source code for all example programs in Appendix A.

In this section we use a few notational shortcuts to make the rules more legible. First, we leave out most domain literals from the rule bodies. We take the approach that the global variables are strictly typed. For example, in first two examples the variable I will have the type *time*/1. The intuition is that there is an implicit literal $\mathit{time}(I)$ in the body of each rule that contains I somewhere in it.

Second, we write a literal set $\rho X.\langle l(X) : a_1(X), \ldots a_m(X)\}\rangle$ as:

$$l(X) : a_1(X) \wedge \cdots \wedge a_m(X)$$

and use the convention that a variable is local only if it occurs in one literal set of the rule and it occurs there in some condition a_i. Otherwise, it is global. Thus, in a rule:

$$1 \leq \{\mathit{hc}(X,Y) : \mathit{edge}(X,Y)\} \leq 1 \leftarrow \mathit{vtx}(X)$$

the variable X is global since it occurs in two places and Y is local as it occurs only in one literal set. Another example is the rule:

$$\leftarrow 2 \leq \{\mathit{push}(X,Y,\mathit{Dir},I) : \mathit{m\text{-}square}(X,Y) \wedge \mathit{direction}(\mathit{Dir})\}$$

Here X, Y, and *Dir* are local variables since they are explicitly mentioned in the conditions of *push*, but I is global since it is not.

[9] Sam Loyd, the originator of the puzzle found a 17-step answer [38]. This author does not know of any previously published 16-step answers.

Variable	Domain literal
L, L_i	*place*(L), *place*(L_i)
S, S_i	*sex*(S), *sex*(S_i)
P, P_i	*person*(P), *person*(P_i)
I	*time*(I)

Table 3: Domain literals of the planning puzzle

10.1 A Planning Puzzle

Consider the following logical puzzle by Sam Loyd [38]:

> It is told that four men eloped with their sweethearts, but in carrying out their plan were compelled to cross a stream in a boat which would hold but two persons at a time. In the middle of the stream, as shown in the sketch, there is a small island. It appears that the young men were so extremely jealous that not one of them would permit his prospective bride to remain at any time in the company of any other man or men unless he was also present.
>
> Nor was any man to get into a boat alone when there happened to be a girl alone, on the island or shore, other than the one to whom he was engaged. This leads one to suspect that the girls were also jealous and feared that their fellows would run off with the wrong girl if they got a chance. Well, be that as it may, the problem is to guess the quickest way to get the whole party across the river.
>
> Let us suppose the river to be two hundred yards wide, with an island in the middle on which any number can stand. How many trips would the boat make to get the four couples safely across in accordance with the imposed conditions?

The first step in solving this puzzle is to define the vocabulary of predicates. As the most important concepts of the puzzle are persons, their locations, and movement, we will construct the problem around the following predicates:

- *at*(L, S, P, I) — the person of sex S from the couple P is at location L at the time step T.
- *boat*(L, I) — the boat is at the location L at the time step T.
- *row*(S, P, I) — the person of sex S from the couple P changes his or her location on the time step T.

The domain definitions for global variables are shown in Table 3.

Conceptually we divide the ω-restricted program into two parts, generator and constraints. The generator part is used to construct all possible plans to get all persons across, and the constraint part prunes out all candidate plans that do not satisfy the conditions of the puzzle.

The core of the generator is two choice rules that are used to generate the possible choices of persons who move at each time step and possible locations of boats:

$$\{\mathit{row}(S,P,I)\} \leftarrow \mathit{at}(L,S,P,I), \mathit{boat}(L,I)$$
$$\{\mathit{boat}(L,I)\} \ .$$

Then, we add to the generator rules stating that all persons who row in the boat end up in the place where the boat is going, and those who do not move stay in the same place:

$$\mathit{at}(L,S,P,I+1) \leftarrow \mathit{row}(S,P,I), \mathit{boat}(L,I+1)$$
$$\mathit{at}(L,S,P,I+1) \leftarrow \mathit{at}(L,S,P,I), \text{not } \mathit{row}(S,P,I) \ .$$

The constraint part can further be divided into two parts. The first part ensures that the world stays consistent and the second part that ensures that the constraints given in the puzzle are satisfied.

First three constraints state that the boat has to be in one place at a time and it has to move on each time step:

$$\leftarrow 2 \leq \{\mathit{boat}(X,I) : \mathit{place}(X)\}$$
$$\leftarrow \{\mathit{boat}(X,I) : \mathit{place}(X)\} \leq 0$$
$$\leftarrow \mathit{boat}(L,I), \mathit{boat}(L,I+1)$$

Since a boat cannot travel empty, we have to add constraints to force at least one and at most two travelers at each time step:

$$\leftarrow 3 \leq \{\mathit{row}(S',P',I) : \mathit{person}(S',P')\}$$
$$\leftarrow \{\mathit{row}(S',P',I) : \mathit{person}(S',P')\} \leq 0$$

Note that we do not need to explicitly state that a person has to be in a single place since our rules for $\mathit{at}/4$ already ensure that.

Next, we start modeling the constraints given in the puzzle. First, no woman may be in the company of men if her own groom is not present:

$$\leftarrow \mathit{at}(L,\mathit{woman},P_1,I), \text{not } \mathit{at}(L,\mathit{man},P_1,I), \mathit{at}(L,\mathit{man},P_2,I), P_1 \neq P_2 \ .$$

Second, no man may row alone if there is some woman other than his fiance alone in any place. To model this constraint we need two additional predicates, $\mathit{has\text{-}company}(S,P,I)$ that is true if the person S-P is not alone at the time step I and $\mathit{two\text{-}move}(I)$ that is true when two persons are moving on the time step I.

$$\leftarrow \mathit{row}(\mathit{man},P_1,I), \text{not } \mathit{two\text{-}move}(I),$$
$$\text{not } \mathit{has\text{-}company}(\mathit{woman},P_2,I), P_1 \neq P_2$$
$$\mathit{two\text{-}move}(I) \leftarrow 2 \leq \{\mathit{row}(P',S',I) : \mathit{person}(P',S')\}$$
$$\mathit{has\text{-}company}(\mathit{woman},P,I) \leftarrow 1 \leq \{\mathit{at}(L,S',P',I) : \mathit{person}(S',P') \wedge P \neq P'\},$$
$$\mathit{at}(L,\mathit{woman},P,I) \ .$$

Finally, we define the initial state and add a constraint to force all persons to end in the correct side of the river:

$$\begin{gathered} \mathit{at}(\mathit{left}, S, P, 1) \\ \leftarrow 1 \leq \{\text{not } \mathit{at}(\mathit{right}, S', P', t+1) : \mathit{person}(S', P')\} \ . \end{gathered}$$

Here t is a 0-ary constant that evaluates to the desired length of the plan. Similarily, p is a constant that evaluates to the number of couples in the puzzle.

The only thing that is left is to define the domain predicates that are used above. They can be defined using the rules:

$$\begin{gathered} \mathit{pair}(1) \leftarrow; \ldots; \mathit{pair}(p) \leftarrow \\ \mathit{time}(1) \leftarrow; \ldots; \mathit{time}(t) \leftarrow \\ \mathit{place}(\mathit{left}) \leftarrow; \quad \mathit{place}(\mathit{right}) \leftarrow; \quad \mathit{place}(\mathit{island}) \leftarrow \\ \mathit{sex}(\mathit{woman}) \leftarrow; \quad \mathit{sex}(\mathit{man}) \leftarrow \\ \mathit{person}(S, P) \leftarrow \mathit{sex}(S), \mathit{pair}(P) \ . \end{gathered}$$

The stable models of the program above correspond to valid solutions of the puzzle. However, the search space of the program is so large, that finding any stable model for it takes many hours using *lparse*-1.0.11 and *smodels*-2.26 on a 1GHz AMD Athlon XP.

If we want to find the optimal solution in a reasonable time, we have to make our program more efficient. We can do this by encoding our knowledge of the problem domain into constraints. For example, we know for certain that in an optimal plan the boat may not have an identical load for two time steps in a row since otherwise the plan could be shortened by leaving the first move out. This constraint can be encoded as:

$$\begin{aligned} \mathit{id\text{-}move}(I) \leftarrow\ & \mathit{row}(S_1, P_1, I), \mathit{row}(S_1, P_1, I+1), \\ & \mathit{row}(S_2, P_2, I), \mathit{row}(S_2, P_2, I+1), \\ & 1 \leq \{S_1 \neq S_2, P_1 < P_2\} \\ \mathit{id\text{-}move}(I) \leftarrow\ & \mathit{row}(S, P, I), \mathit{row}(S, P, I+1), \\ & \text{not } \mathit{two\text{-}move}(I), \text{not } \mathit{two\text{-}move}(I+1) \end{aligned}$$

The constraint literal $1 \leq \{S_1 \neq S_2, P_1 < P_2\}$ is true when the two persons either belong to a different couple or they are of different sex. The use of the $<$-relation is a further optimization that ensures that only one rule is created for each pair of persons.

Another optimization that can be done is to note that it is not possible for two persons that are of different sex and belong to different couples to move at the same time:

$$\leftarrow \mathit{row}(\mathit{woman}, P_1, I), \mathit{row}(\mathit{man}, P_2, I), P_1 \neq P_2 \ .$$

Perhaps the most important way of reducing complexity in this case is to reduce symmetries in it. In the puzzle each couple is identical in the sense that they all follow the same rules. If we have a valid solution, then systematically swapping all moves of persons in couples i and j together results in

n	Len	Sol	Basic encoding			With constraints			Optimized		
			#At	#Rl	Time	#At	#Rl	Time	#At	$\#Rl$	Time
1	1	Y	43	41	0 s	34	41	0.01 s	43	50	0 s
2	5	Y	192	367	0.02 s	201	468	0.02 s	232	506	0.01 s
	4	N	157	295	0.01 s	164	374	0 s	190	404	0 s
3	11	Y	540	1241	5.29 s	583	1830	0.48 s	646	1915	0.33 s
	10	N	493	1129	120.86 s	532	1663	0.52 s	590	1740	0.59
4	16	Y	973	2572	—	1048	4192	130.79 s	1174	4320	46.06 s
	15	N	914	2412	—	1031	3931	165.44 s	1103	4049	156.41

Table 4: A comparison of different encodings of the puzzle

Move	Left	Island	Right
1	BCDbcd	Aa+	
2	ABCDbcd+	a	
3	ABCDd	a	bc+
4	ABCDbd+	a	c
5	BDbd	a	ACc+
6	ABDbd+	a	Cc
7	ADd	a	BCbc+
8	ADd	ab+	BCc
9	ADad+	b	BCc
10	ad	b	ABCDc+
11	ad	Bb+	ACDc
12	ad		ABCDbc+
13	abd+		ABCDc
14	d		ABCDabc+
15	Dd+		ABCabc
16			ABCDabcd+

Table 5: A balanced optimal solution for the 4-elopers puzzle

another valid plan. It is possible that during stable model computation bits of these plans get mixed; each subplan is a part of a consistent answer but together they yield a contradiction. If we remove as many of these symmetries as we can by adding suitable constraints, then we are likely to get answers much faster than with them.

We define that two couples are equivalent if both men and both women are in the same places. Then, we demand that persons of the lower numbered couple have to move first.

$$\begin{aligned} \mathit{equivalent}(P_1, P_2, I) \leftarrow\ & \mathit{at}(L_1, S_1, P_1, I), \mathit{at}(L_1, S_1, P_2, I), \\ & \mathit{at}(L_2, S_2, P_1, I), \mathit{at}(L_2, S_2, P_2, I), \\ & S_1 \neq S_2, P_1 < P_2 \\ \leftarrow\ & \mathit{row}(S, P_2, I), \text{not } \mathit{row}(S, P_1, I), \\ & \mathit{equivalent}(P_1, P_2, I) \end{aligned}$$

After adding these constraints we could find an optimal plan in 118 seconds on the same computer system. Proving optimality takes about twice longer, 243 s. We also present a further optimized version of the program in Appendix A.2. With this encoding, the optimal solution can be found in 46 s and the optimality proven in 156 s. We compared the three different encodings also in cases where there are less than 4 couples. These results are presented in Table 4.

One optimal solution for the puzzle is shown in Table 5. The men are denoted using capital and the women with lower case letters. The location of the boat is marked using the +-symbol. This solution is balanced in the sense that both men and women move as many times.

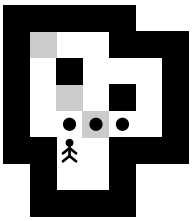


Figure 14: A Sokoban game instance

10.2 Sokoban

In the game of Sokoban, the player takes the role of a warehouse keeper ("Sokoban") who has to rearrange a number of boxes. The 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 puzzle[10] is shown in Figure 14.

The first step in modeling Sokoban is to decide what a move will be. One possibility would be to use the Sokoban's movement directly, so that one step of the Sokoban is one move. 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 the 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.

In the encoding of the Sokoban problem we use extended cardinality constraint programs, that is, we use classical negation in some rules. The most important predicates in use are:

- *has-box*(X, Y, I) — there is a box at location (X, Y) at the time step I;
- *push*(X, Y, Dir, I) — the Sokoban pushes the box that is at (X, Y) to the direction *Dir* at the time step I;
- *can-push*(X, Y, Dir, I) — the box that is at (X, Y) can be pushed to the direction *Dir* at the time step I;
- *move-to*(X, Y, I) — a box is pushed to (X, Y) at the time step I;
- *reachable*(X, Y, I) — the Sokoban can reach (X, Y) at the time step I; and
- *at*(X, Y, I) — the Sokoban is at (X, Y) at the time step I.

When defining the domain of a puzzle, we classify the possible locations of the warehouse into two different classes:

- *square*(X, Y) — there is an open space at location (X, Y); and

[10] This level is designed by Yoshio Murase and it is downloaded from `http://www.ne.jp/asahi/ai/yoshio/sokoban/main.htm`.

Variable	Domain literal
$X, Y; X_i, Y_i$	*m-square*(X, Y), *m-square*(X_i, Y_i)
Dir	*direction*(*Dir*)
I	*time*(I)

Table 6: Domain literals of Sokoban

- *m-square*(X, Y) — there is an open space at location (X, Y) such that it is possible to push a box from (X, Y) to one of the target squares.

We use the predicate *m-square*/2 to reduce the number of possible rules. For example, if we push a box into a corner (x, y), then there is no way to move it again and such a move cannot belong to a valid plan. In this case *square*(x, y) would be true but *m-square*(x, y) would not. The predicate *m-square*/2 is a domain predicate that is defined using recursive rules.

As was the case in the previous section, we divide the program into two parts, one of which generates all possible plans, and one that ensures that all constraints of the problem are satisfied.

The core of the generator is again formed by two choice rules, one that selects which box is pushed and another that selects the target:

$$\begin{aligned} \{\textit{push}(X, Y, \textit{Dir}, I)\} \leftarrow\ & \textit{has-box}(X, Y, I), \textit{can-push}(X, Y, \textit{Dir}, I), \\ & \textit{has-neighbor}(X, Y, \textit{Dir}) \\ 1 \leq \{\textit{move-to}(X_2, Y_2, I) : \textit{same-segment}(X_1, Y_1, X_2, Y_2, \textit{Dir})\} \leq 1 \leftarrow\ & \\ & \textit{push}(X_1, Y_1, \textit{Dir}, I), \textit{has-neighbor}(X_1, Y_1, \textit{Dir}) \end{aligned}$$

Here the predicates in the rule bodies abstract much of the complexity of the problem domain. The predicate *can-push*$(X, Y, \textit{Dir}, I)$ is true if the Sokoban can push the box in (X, Y) to *Dir* at I. The predicate *has-neighbor*/3 is a domain predicate that is true if there is possibly an empty space to the direction *Dir* from (X, Y).

The predicate *same-segment*$(X_1, Y_1, X_2, Y_2, \textit{Dir})$ is a domain predicate that encodes the fact that (X_1, Y_1) and (X_2, Y_2) are on the same straight line and there are no structural obstructions between them.

Next, we add a rule to say that we can move only one box at a time, and the box may not be pushed over or onto another:

$$\begin{aligned} &\leftarrow 2 \leq \{\textit{push}(X, Y, \textit{Dir}, I) : \textit{m-square}(X, Y) \wedge \textit{direction}(\textit{Dir})\} \\ &\leftarrow \textit{push}(X_1, Y_1, \textit{Dir}, I), \textit{move-to}(X_3, Y_3, I), \textit{has-box}(X_2, Y_2, I), \\ &\quad \textit{same-segment}(X_1, Y_1, X_2, Y_2, \textit{Dir}), \textit{same-segment}(X_2, Y_2, X_3, Y_3, \textit{Dir}) \\ &\leftarrow \textit{has-box}(X, Y, I), \textit{move-to}(X, Y, I)\ . \end{aligned}$$

A box that is pushed ends up in the new place, and a box that is not moved stays in place:

$$\begin{aligned} \textit{has-box}(X, Y, I+1) &\leftarrow \textit{move-to}(X, Y, I) \\ \neg\textit{has-box}(X, Y, I+1) &\leftarrow \textit{push}(X, Y, \textit{Dir}, I) \\ \textit{has-box}(X, Y, I+1) &\leftarrow \text{not } \neg\textit{has-box}(X, Y, I+1), \textit{has-box}(X, Y, I) \end{aligned}$$

Next, we define when the Sokoban can push boxes. Here we need four rules, one for each direction. All rules share the same form:

$$\begin{aligned} \textit{can-push}(X, Y, \textit{east}, I) \leftarrow\ & \textit{has-box}(X, Y, I), \text{not } \textit{has-box}(X+1, Y, I), \\ & \textit{reachable}(X-1, Y, I), \textit{square}(X-1, Y) \\ & \textit{m-square}(X, Y), \textit{m-square}(X+1, Y) \ . \end{aligned}$$

The predicate *reachable*/3 denotes all locations that the Sokoban can reach at a given time step. It is computed as a transitive closure of unblocked squares that are adjacent to the Sokoban:

$$\begin{aligned} \textit{reachable}(X, Y, I) \leftarrow\ & \textit{square}(X, Y), \textit{at}(X, Y, I) \\ \textit{reachable}(X+1, Y, I) \leftarrow\ & \textit{reachable}(X, Y, I), \text{not } \textit{has-box}(X'+1, Y', I) \\ & \textit{square}(X, Y), \textit{square}(X+1, Y) \ . \end{aligned}$$

Similar rules are needed also for the other three directions. We explicitly mention the domain literals *square*/2 in the bodies of these rules to emphasize that the Sokoban can himself move to places where he should not push boxes into.

Before defining the domain predicates, we have to add a rule that explains where the Sokoban is after a move:

$$\textit{at}(X, Y, I+1) \leftarrow \textit{push}(X, Y, \textit{Dir}, I) \ .$$

In defining the domain predicates we once again have to use transitive closure in defining some recursive relations. First, we define the predicates *square*/2 and *target-square*/2 directly as facts according to the given problem instance. Based on this predicate we define the predicate *m-square*/2 to be the set of squares from where it is possible to push a box to some target location. This rules out, for example, squares in a corner since if a box is pushed to the corner, it may never be retrieved from it.

$$\begin{aligned} \textit{m-square}(X, Y) \leftarrow\ & \textit{square}(X, Y), \textit{has-target-route}(X, Y) \\ \textit{has-target-route}(X, Y) \leftarrow\ & \textit{target-square}(X, Y) \\ \textit{has-target-route}(X+1, Y) \leftarrow\ & \textit{has-target-route}(X, Y), \textit{square}(X, Y), \\ & \textit{square}(X+1, Y), \textit{square}(X-1, Y) \ . \end{aligned}$$

In the third rule the atom $\textit{square}(X-1, Y)$ was necessary so that we can ensure that there is really space to push the box to the correct direction. Obviously, we again need four rules of this form, one for each direction.

The final missing domain predicate is *same-segment*/4 that again has to be defined recursively using eight rules of the form:

$$\begin{aligned} \textit{same-segment}(X, Y, X+1, Y, \textit{east}) \leftarrow\ & \textit{m-square}(X, Y), \\ & \textit{m-square}(X, Y). \\ \textit{same-segment}(X_1, Y, X_2+1, Y, \textit{east}) \leftarrow\ & \textit{same-segment}(X_1, Y, X_2, Y, \textit{east}), \\ & \textit{m-square}(X_1, Y), \\ & \textit{m-square}(X_2, Y), \\ & \textit{m-square}(X_2+1, Y) \ . \end{aligned}$$

Our goal state is such that all target squares have to have a box in the final step:

$$\leftarrow 1 \leq \{\text{not } \textit{has-box}(X, Y, t+1) : \textit{target-square}(X, Y)\} \ .$$

As in the planning puzzle example, we can make the program more efficient by adding certain constraints to prune out incorrect branches of the search tree. For example, the Sokoban should never push two boxes together along a wall since they will be stuck there. This constraint can be encoded as:

$$\begin{aligned}
\textit{edge-pair}(X, Y, X+1, Y) \leftarrow\ & \text{not } \textit{target-square}(X, Y), \\
& \text{not } \textit{target-square}(X+1, Y), \\
& \textit{square}(X, Y), \textit{square}(X+1, Y), \\
& \text{not } \textit{square}(X, Y-1), \\
& \text{not } \textit{square}(X+1, Y-1) \ . \\
\leftarrow\ & \textit{edge-pair}(X_1, Y_1, X_2, Y_2), \\
& \textit{has-box}(X_1, Y_1, I), \textit{has-box}(X_2, Y_2, I) \ .
\end{aligned}$$

Similar rules may be used to forbid cases where three boxes are pushed to an L-shape around a corner or four boxes are pushed together to form a square.

There are also optimizations that make candidate solutions shorter. For example, a box should not be pushed twice to the same direction:

$$\begin{aligned}
\textit{push-dir}(\textit{Dir}, I) \leftarrow\ & \textit{push}(X, Y, \textit{Dir}, I) \\
\leftarrow\ & \textit{push-dir}(\textit{Dir}, I), \textit{move-to}(X, Y, I), \\
& \textit{push}(X, Y, \textit{Dir}, I+1) \ .
\end{aligned}$$

This program can be used to find the solution for the puzzle instance shown in Figure 14. An optimal 10-step solution is shown in Figure 15.

10.3 Creating Finite Automata

A *deterministic finite automaton* (DFA) is a five-tuple $M = \langle Q, \Sigma, \delta, s, F \rangle$ where Q is a finite set of states, Σ a finite alphabet, s the initial state, $F \subseteq Q$ a set of accepting states, and a transition function $\delta : Q \times \Sigma \rightarrow Q$ that associates a new state to each pair of a state and input symbol. A *non-deterministic finite automaton* (NFA) is otherwise similar but instead of a transition function there is a transition relation $\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$. Each nondeterministic automaton can be translated into a possibly exponentially larger deterministic automaton using the well-known subset construction (see, for example, [32]).

In this section we consider the problem of creating finite automata that satisfy certain structural properties. When teaching a basic course on theoretical computer science, there is a need for a large number of automata theoretic exercises especially if each student should have exercises that are unique for him or her. Even though there are many published exercises in the numerous text books on the subject, they are likely to run out if the number of students on the courses reaches several hundreds. We would like to generate a number of exercises that are not too easy or too difficult and that are all roughly equally difficult.

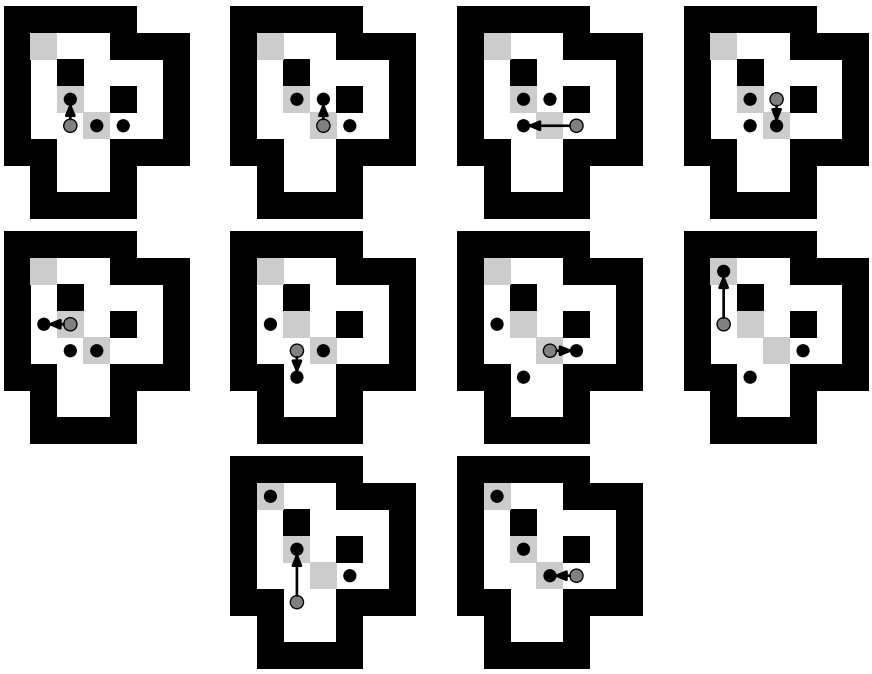


Figure 15: Solution of the Sokoban instance

Here we concentrate on the problem of determinising a finite automaton. In practice, we want to generate a number of non-deterministic finite automata M such that determinising them leads to interesting deterministic automata. We are interested in the following parameters of automata:

- n — the number of states in the NFA;
- min_d, max_d — the minimum and the maximum number of states in a corresponding DFA;
- min_f, max_f — the minimum and the maximum number of accepting states in a corresponding DFA;
- min_E, max_E — the minimum and the maximum number of edges in the NFA;
- min_σ — the minimum number of transitions in NFA for each symbol of the alphabet Σ;
- $min_\varepsilon, max_\varepsilon$ — the minimum and the maximum number of empty ε-transitions in the NFA; and
- min_a, max_a — the minimum and maximum number of states in the NFA that have two or more transitions sharing the same input symbol.

In the first part of the program we nondeterministically generate a NFA. We start by defining the states and the alphabet of the NFA. We will use $\Sigma = \{a, b\}$ as the alphabet.

$$\begin{aligned} \mathit{nd\text{-}state}(1) \leftarrow; \cdots; \mathit{nd\text{-}state}(n) &\leftarrow \\ \mathit{symbol}(a) \leftarrow; \quad \mathit{symbol}(b) &\leftarrow; \\ \mathit{nd\text{-}symbol}(S) &\leftarrow \mathit{symbol}(S) \\ \mathit{nd\text{-}symbol}(\varepsilon) &\leftarrow \quad . \end{aligned}$$

Next, we choose the sets of transitions and final states of the NFA:

$$\{\textit{nd-transition}(Q_1, S, Q_2)\} \leftarrow \textit{nd-state}(Q_1), \textit{nd-state}(Q_2), \textit{nd-symbol}(S)$$
$$\{\textit{nd-final}(Q)\} \leftarrow \textit{nd-state}(Q) \ .$$

The transitions of the NFA have to fullfill the given parameters:

$$\begin{aligned}
\textit{ok}_E &\leftarrow \textit{min}_E \leq \{\textit{nd-transition}(Q_1, S, Q_2) : \textit{nd-state}(Q_1) \wedge \\
&\qquad \textit{nd-symbol}(S) \wedge \textit{nd-state}(Q_2)\} \leq \textit{max}_E \\
\textit{ok}_\varepsilon &\leftarrow \textit{min}_\varepsilon \leq \{\textit{nd-transition}(Q_1, \varepsilon, Q_2) : \textit{nd-state}(Q_1) \wedge \\
&\qquad \textit{nd-state}(Q_2)\rangle\} \leq \textit{max}_\varepsilon \\
\textit{ok}_a &\leftarrow \textit{min}_a \leq \{\textit{has-two}(Q) : \textit{nd-state}(Q)\} \leq \textit{max}_a \\
\textit{has-two}(Q_1) &\leftarrow 2 \leq \{\textit{nd-transition}(Q_1, S, Q_2) : \textit{nd-state}(Q_2)\}, \\
&\qquad \textit{nd-symbol}(S), \textit{nd-state}(Q_1) \\
\textit{ok}_\sigma &\leftarrow \textit{min}_\sigma \leq \{\textit{nd-transition}(Q_1, S, Q_2) : \textit{nd-state}(Q_1) \wedge \\
&\qquad \textit{nd-state}(Q_2)\}, \textit{symbol}(S) \\
&\leftarrow 1 \leq \{\text{not } \textit{ok}_\varepsilon, \text{not } \textit{ok}_a, \text{not } \textit{ok}_E, \text{not } \textit{ok}_\Sigma, \text{not } \textit{ok}_d\} \ .
\end{aligned}$$

Additionally, we want to impose some sanity constraints for the NFA. Namely, every state of the automaton should be reachable from the initial state, and there should not states with no path to an accepting state.

$$\begin{aligned}
\textit{reachable}(1) &\leftarrow \\
\textit{reachable}(Q_2) &\leftarrow \textit{nd-transition}(Q_1, S, Q_2), \textit{reachable}(Q_1), \\
&\qquad \textit{nd-state}(Q_1), \textit{nd-state}(Q_2), \textit{symbol}(S) \\
&\leftarrow \textit{nd-state}(Q), \text{not } \textit{reachable}(Q) \\
\textit{accepts}(Q) &\leftarrow \textit{nd-final}(Q), \textit{nd-state}(Q) \\
\textit{accepts}(Q_1) &\leftarrow \textit{nd-transition}(Q_1, S, Q_2), \textit{accepts}(Q_2), \\
&\qquad \textit{nd-state}(Q_1), \textit{nd-state}(Q_2), \textit{symbol}(S) \\
&\leftarrow \textit{nd-state}(Q), \text{not } \textit{accepts}(Q)
\end{aligned}$$

Finally, every symbol in alphabet should occur in the machine, no state should have only a single ε-transition leading out from it, and there should be no empty self-loops:

$$\begin{aligned}
&\leftarrow \{\textit{nd-transition}(Q_1, S, Q_2) : \textit{nd-state}(Q_1) \wedge \textit{nd-state}(Q_2)\} \leq 0, \\
&\quad \textit{symbol}(S) \\
&\leftarrow \{\textit{nd-transition}(Q_1, \varepsilon, Q_2) : \textit{nd-state}(Q_2)\} \leq 1, \\
&\quad \{\textit{nd-transition}(Q_1, S, Q_3) : \textit{nd-state}(Q_3) \wedge \textit{nd-symbol}(S)\} \leq 0, \\
&\quad \textit{nd-state}(Q_1) \\
&\leftarrow \textit{nd-transition}(Q, \varepsilon, Q), \textit{nd-state}(Q) \ .
\end{aligned}$$

Next, we want to determinize the NFA that was generated by the above rules. Each state of the DFA corresponds to a set of states of the NFA. Determinizing an n-state NFA may result in a 2^n state DFA so we will encode the states of the DFA using n bit natural numbers. The idea is that if the ith bit

of the DFA state q is true, then the state q_i of the NFA belongs to the state set of q. We model this relation using the predicate $\mathit{in}(Q_n, Q_d)$ that is true exactly when $q_n \in q_d$. The state predicates are defined explicitly by the set of facts:

$$\{\mathit{in}(i,j) \leftarrow | \text{ the } j\text{th bit of } i \text{ is } 1\} \cup \{\mathit{d\text{-}state}(i) \leftarrow i \in [0, 2^n - 1]\}$$

When we create the states of the DFA, we have to compute the ε-closures of the states of the NFA:

$$\begin{aligned}
\mathit{closure}(Q, Q) &\leftarrow \mathit{nd\text{-}state}(Q) \\
\mathit{closure}(Q_1, Q_2) &\leftarrow \mathit{nd\text{-}transition}(Q_1, \varepsilon, Q_2), \mathit{nd\text{-}state}(Q_1), \mathit{nd\text{-}state}(Q_2) \\
\mathit{closure}(Q_1, Q_3) &\leftarrow \mathit{nd\text{-}transition}(Q_2, \varepsilon, Q_3), \mathit{closure}(Q_1, Q_2), \\
&\qquad \mathit{nd\text{-}state}(Q_1), \mathit{nd\text{-}state}(Q_2), \mathit{nd\text{-}state}(Q_3) \ .
\end{aligned}$$

The initial state of the DFA contains the whole ε-closure of the initial state of the NFA and nothing more. Here the definition is simpler if we use two auxiliary predicates: *not-initial*/1 to denote the DFA states that contain NFA states that do not belong to the ε-closure of the initial state and *impossible*/1 to denote the DFA states that are not ε-closed:

$$\begin{aligned}
\mathit{d\text{-}initial}(Q) &\leftarrow \mathit{d\text{-}state}(Q), \mathit{in\text{-}state}(1, Q), \\
&\qquad \text{not } \mathit{impossible}(Q), \\
&\qquad \text{not } \mathit{not\text{-}initial}(Q) \\
\mathit{not\text{-}initial}(Q) &\leftarrow \mathit{in\text{-}state}(N, Q), \text{not } \mathit{closure}(1, N) \\
\mathit{impossible}(Q) &\leftarrow \mathit{in\text{-}state}(N_1, Q), \text{not } \mathit{in\text{-}state}(N_2, Q), \mathit{closure}(N_1, N_2), \\
&\qquad \mathit{d\text{-}state}(Q), \mathit{nd\text{-}state}(N_1), \mathit{nd\text{-}state}(N_2) \ .
\end{aligned}$$

There is a transition from a ε-closed DFA state q_i to q_j with symbol a if q_j contains the ε-closure of all states q' such that there is a NFA transition $(q_i, a, q') \in \Delta$. The simplest way to define this relation with rules is via its complement; assume by default that all transitions are possible, but remove all incorrect transitions:

$$\begin{aligned}
\mathit{d\text{-}transition}(Q_1, S, Q_2) \leftarrow &\text{ not } \mathit{impossible}(Q_2), \text{not } \mathit{wrong}(Q_1, S, Q_2) \\
&\mathit{d\text{-}state}(Q_1), \mathit{d\text{-}state}(Q_2), \mathit{symbol}(S), \\
&\mathit{d\text{-}reachable}(Q_1) \ .
\end{aligned}$$

The predicate *d-reachable*/1 is used to simplify the resulting DFA by removing all unreachable states and their transitions from it. The predicate *wrong*/3 is defined as follows:

$$\begin{aligned}
\mathit{wrong}(Q_1, S, Q_2) &\leftarrow \mathit{in\text{-}state}(N_1, Q_1), \text{not } \mathit{in\text{-}state}(N_2, Q_2), \\
&\qquad \mathit{nd\text{-}transition}(N_1, S, N_2), \mathit{symbol}(S) \\
\mathit{wrong}(Q_1, S, Q_2) &\leftarrow \mathit{in\text{-}state}(N, Q_2), \text{not } \mathit{has\text{-}transition}(Q_1, S, N), \\
&\qquad \mathit{d\text{-}state}(Q_1), \mathit{symbol}(S)
\end{aligned}$$

The predicate $\mathit{has\text{-}transition}(Q_1, S, N)$ encodes the fact that there is some way to reach from some state belonging in Q_1 to the NFA state N while

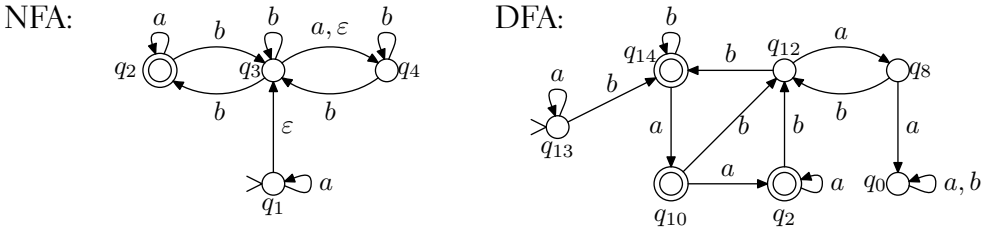


Figure 16: A NFA and the corresponding DFA

reading the symbol S from input:

$$\begin{aligned}
\mathit{has\text{-}transition}(Q, S, N_2) \leftarrow\ & \mathit{in\text{-}state}(N_1, Q), \mathit{nd\text{-}transition}(N_1, S, N_2), \\
& \mathit{nd\text{-}state}(N_2), \mathit{symbol}(S) \\
\mathit{has\text{-}transition}(Q, S, N_2) \leftarrow\ & \mathit{has\text{-}transition}(Q, S, N_1), \mathit{closure}(N_1, N_2), \\
& \mathit{q\text{-}state}(Q), \mathit{symbol}(S), \\
& \mathit{nd\text{-}state}(N_1), \mathit{nd\text{-}state}(N_2) \ .
\end{aligned}$$

We find the set of reachable DFA states by computing the transitive closure of the transition function:

$$\begin{aligned}
\mathit{d\text{-}reachable}(Q) \leftarrow\ & \mathit{d\text{-}initial}(Q), \mathit{d\text{-}state}(Q) \\
\mathit{d\text{-}reachable}(Q_2) \leftarrow\ & \mathit{d\text{-}transition}(Q_1, S, Q_2), \mathit{d\text{-}state}(Q_1), \\
& \mathit{d\text{-}state}(Q_2), \mathit{symbol}(S) \ .
\end{aligned}$$

Finally, a DFA state is accepting if some NFA state belonging to it is:

$$\mathit{d\text{-}final}(Q) \leftarrow \mathit{d\text{-}reachable}(Q), \mathit{in\text{-}state}(N, Q), \mathit{nd\text{-}final}(N) \ .$$

Finally, we want to ensure that the resulting DFA has a correct number of states:

$$\begin{aligned}
\mathit{ok}_d \leftarrow\ & \mathit{min}_d \leq \{\mathit{d\text{-}reachable}(Q) : \mathit{d\text{-}state}(Q)\} \leq \mathit{max}_d \\
\mathit{ok}_f \leftarrow\ & \mathit{min}_f \leq \{\mathit{d\text{-}final}(Q) : \mathit{d\text{-}state}(Q)\} \leq \mathit{max}_f \\
\leftarrow\ & 1 \leq \{\text{not } \mathit{ok}_d, \text{not } \mathit{ok}_f\} \ .
\end{aligned}$$

As the size of the ground instantiation of the program is always exponential to the number of states in the NFA and computing a stable model is **NP**-complete with respect to size of the instantiation, we can expect it to handle only small problem instances. In practice, this is not a severe restriction since at least the first exercises on a basic course should be quite small.

The Figure 16 shows a NFA and the corresponding DFA created with the program. The parameters were:

$$\begin{array}{cccc}
n = 4 & \mathit{min}_\sigma = 2 & \mathit{min}_d = 7 & \mathit{max}_d = 9 \\
\mathit{min}_E = 6 & \mathit{max}_E = 20 & \mathit{min}_f = 2 & \mathit{max}_f = 3 \\
\mathit{min}_\varepsilon = 2 & \mathit{max}_\varepsilon = 3 & \mathit{min}_a = 2 & \mathit{max}_a = 4 \ .
\end{array}$$

11 CONCLUSIONS AND FUTURE WORK

In this work we defined the stable model semantics for cardinality constraint programs. The semantics of a program is defined with respect to a data model. The basic idea is that all rules with variables are replaced by their ground instantiations. The instantiation is done in two steps: first the global variables in a rule are instantiated, and then the local variables that occur in literal sets are expanded.

We also defined ω-restricted programs that allow a programmer to define the data model using inference rules. The basic idea is to construct a hierarchy of predicate symbols occurring in the program so that more complex data predicates are defined in terms of simpler predicates. The ω-restricted programs have the property that they stay decidable even if function symbols are allowed. The ω-restricted programs have been implemented in the SMODELS system.

We gave a framework for extending the stable model semantics for different types of constraint literals and as an example defined four different semantics: cardinality constraints with variables in bounds, weight constraint literals, classical negation, and partial model semantics.

We showed how some of these semantics can be implemented using standard cardinality constraint rules and also showed how the ordered disjunction semantics can be implemented using a dual-program construction.

We analyzed the computational complexity and found that in the general case ω-restricted programs are 2-**NEXP**-complete, and **NEXP**-complete if function symbols are not allowed.

Finally, we presented a few implementation details of the SMODELS system as well as three larger programming examples where ω-restricted programs were used to solve practical problems.

11.1 Future Work

This work leaves room for further work on several different areas of cardinality constraint programs. For example, the formal definitions for data models presented in Section 4 are complex and it may be possible to find a simpler way to explain the necessary concepts.

Another interesting area is identifying and analyzing different data models that define different types of built-in functions and predicates. For example, the SMODELS system contains built-in range definitions of the form

$$d(1 \,..\, n) \leftarrow$$

where n may be any term evaluating to an integer. This rule defines the set of $n + 1$ facts:

$$\{d(i) \leftarrow \mid 1 \leq i \leq n\} \ .$$

It is clear that range definitions cause significant increase in computational complexity but it is not at all clear how difficult the resulting programs are to solve.

One further possible area of research is to investigate different kinds of constraint literals. In some cases we might want to compute aggregates like

averages, minima, and maxima of some arguments of a set of literals. For example, if we model the behavior of some shopping agent we might want to know if the highest-priced item on our shopping list costs more than some predetermined limit. We could express that using syntax like:

$$\textit{expensive} \leftarrow \max(M, 2)\ \{\textit{buy}(X, P) : \textit{price(P)} \wedge \textit{in-list}(X)\},\ \textit{limit}(L), M > L$$

where $\max(M, 2)$ denotes that M should get the maximum value occurring in the second argument of the literals in the set.

A fourth avenue of research is to provide translations from various logic programming semantics into cardinality constraint programs. This provides a method for quick prototyping logic program implementations. Cardinality constraint programs are quite expressive and you often can create translations along the line of those presented in Section 7 that can easily implemented and so the properties of the different semantics can be examined without having to create a complete implementation from the scratch.

In conclusion, we see that the subject of cardinality constraints and similar constructs still contains many topics suitable for future research.

ACKNOWLEDGEMENTS

I am thankful to Professor Ilkka Niemelä who gave me an opportunity to work in the Laboratory for Theoretical Computer Science and who instructed me during this work. I am also grateful to Docent Tomi Janhunen for his helpful comments on this work.

This work has been financed by Helsinki Graduate School in Computer Science and Engineering as well as by the Academy of Finland (project 53695).

REFERENCES

[1] C. Anger, K. Konczak, and T. 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.

[2] K. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19–20:9–71, 1994.

[3] Y. Babovich and M. Maratea. Cmodels-2: SAT-based answer set solver enhanced to non-tight programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04), to appear*, Miami, Florida, January 2004.

[4] P. A. Bonatti. Resoning with infinite stable models. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 603–610, August 2001.

[5] G. Brewka. Adding priorities and specificity to default logic. In *Logics in Artificial Intelligence, European Workshop, JELIA '94*, pages 247–260, York, UK, 1994.

[6] G. Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4:19–36, 1996.

[7] G. Brewka. Logic programming with ordered disjunction. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI-02)*, pages 100–105. Morgan Kaufmann, 2002.

[8] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109:297–356, 1999.

[9] G. Brewka and T. Eiter. Prioritizing default logic. In *Intellectics and Computational Logic*, pages 27–45, 2000.

[10] G. Brewka, I. Niemelä, and T. 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)*, pages 444–455, 2002.

[11] F. Buccafurri, N. Leone, and P. 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.

[12] P. Cholewiński, V. W. Marek, and M. Truszczyński. Default reasoning system DeReS. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 518–528. Morgan Kaufmann, San Francisco, California, 1996.

[13] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374–425, 2001.

[14] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[15] J. P. Delgrande, T. Schaub, and H. 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.

[16] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System description: Dlv. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 424 – 428, Vienna, Austria, September 2001. Springer-Verlag.

[17] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181, 1997.

[18] D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Advances in Artificial Intelligence, Joint German/Austrian Conference on AI (KI-2001)*, pages 138–153, 2001.

[19] T. Eiter, W. Faber, G. Pfeifer, and N. 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.

[20] T. Eiter and G. Gottlob. Propositional circumscription and extended closed-world reasoning are Π_2^P-complete. *Theoretical Computer Science*, 114:231–245, 1993.

[21] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[22] Eiter, T., Leone, N., Pfeifer G., Mateis C., and Scarcello, F. The kr system `dlv`: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[23] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using database optimization techniques for nonmonotonic reasoning. In *Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pages 135–139, 1999.

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

[25] M. Gelfond and V. 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.

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

[27] M. 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.

[28] K. 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.

[29] T. 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.

[30] T. Janhunen, I. Niemelä, P. Simons, and J.-H. 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.

[31] N. Leone, S. Perri, and F. Scarcello. Improving asp instantiators by join-ordering methods. In *Proceedings of the 6th International Conference Logic Programming and Nonmonotonic Reasoning (LPNMR'01)*, pages 280–294, Vienna, Austria, September 2001.

[32] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation (2nd ed)*. Prentice Hall, 1998.

[33] V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 25–37, Las Cruces, New Mexico, December 1999. The MIT Press.

[34] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.

[35] V. Lifschitz and H. Turner. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 23–37, 1994.

[36] F. Lin and Y. 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.

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

[38] S. Loyd. *Mathematical Puzzles of Sam Loyd*. Dover, 1959.

[39] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. cs.LO/9809032.

[40] V. W. Marek and J. 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.

[41] V. W. Marek and J. B. Remmel. On logic programs with cardinality constraints. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 219–228, 2002.

[42] V. W. Marek and J. B. Remmel. Set constraints in logic programming, 2002. Unpublished draft.

[43] V. W. Marek and M. Truszczyński. *Nonmonotonic Logic*. Springer-Verlag, 1993.

[44] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398. Springer-Verlag.

[45] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.

[46] I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, 1998.

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

[48] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[49] I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429, Dagstuhl, Germany, July 1997. Springer-Verlag.

[50] I. Niemelä and P. Simons. Extending the smodels system with cardinality and weight constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

[51] I. Niemelä, P. Simons, and T. 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.

[52] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc, 1994.

[53] T. C. Przymusinski. Stationary semantics for disjunctive logic programs. In *Proceedings of the North American Logic Programming Conference*, pages 40–59, Austin, Texas, 1990. MIT Press.

[54] T. C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing Journal*, 9(3):401–424, 1991.

[55] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[56] J. Rintanen. *Lexicographic Ordering as a Basis of Priorities in Default Reasoning*. PhD thesis, Helsinki University of Technology, Finland, 1997.

[57] J. Rintanen. Lexicographic priorities in default logic. *Artificial Intelligence*, 106(2):221–265, 1998.

[58] C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1994.

[59] C. Sakama and K. Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artificial Intelligence*, 123(1-2):185–222, 2000.

[60] T. Schaub and K. Wang. A comparative study of logic programs with preference. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 597–602, Seattle, Washington, USA, 2001.

[61] R. Sedgewick. *Algorithms in C*. Addison-Wesley Publishing Company, Inc, 1990.

[62] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report 47, Helsinki University of Technology, Helsinki, Finland, August 1997.

[63] P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag.

[64] P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.

[65] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

[66] T. Soininen. *An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[67] T. Soininen and I. Niemelä. Formalizing configuration knowledge using rules with choices. Technical Report TKO-B142, Laboratory of Information Processing Science, Helsinki University of Technology, 1998.

[68] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, pages 305–319. Springer-Verlag, January 1999.

[69] T. Soininen, I. Niemelä, J. Tiihonen, and R. 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.

[70] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B 18, Helsinki University of Technology, Helsinki, Finland, October 1998.

[71] T. 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.

[72] T. Syrjänen. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic*, pages 837–851, London, UK, July 2000. Springer-Verlag.

[73] T. Syrjänen. Modelling the game of life using logic programs. In N. Husberg, T. Janhunen, and I. Niemelä, editors, *Leksa Notes in Computer Science, Festschrift in Honour of Professor Leo Ojala*, pages 115–124. 2000.

[74] T. Syrjänen. Optimizing configurations. In *Proceedings of the ECAI Workshop W02 on Configuration*, pages 85–90, Berlin, Germany, August 2000.

[75] T. 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.

[76] T. Syrjänen. Version spaces and rule-based configuration management. In *Working notes of the IJCAI 2001 Workshop on Configuration*, August 2001.

[77] T. Syrjänen and I. Niemelä. The smodels system. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, Vienna, Austria, September 2001. Springer-Verlag.

[78] T. Syrjänen. *Lparse User's Manual*. Available at: <url:http://www.tcs.hut.fi/Software/smodels>.

[79] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.

[80] A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery*, 38(3):620–650, July 1991.

A SOURCE CODE FOR EXAMPLE PROGRAMS

Here we give the complete *smodels* source codes for the example problems of Section 10.

A.1 Planning Puzzle

```
pair(1..p).
time(1..t).
place(left; right; island).
sex(woman; man).
person(S, P) :- sex(S), pair(P).

{ row(S, P, I) } :-
    at(From, S, P, I),
    boat(From, I),
    time(I),
    place(From),
    person(S, P).

{ boat(L, I) : place(L) } :- time(I).
boat(right, t+1).

at(To, S, P, I+1) :-
    row(S, P, I),
    boat(To, I+1),
    time(I),
    place(To),
    person(S, P).

at(X, S, P, I+1) :-
    at(X, S, P, I),
    not row(S, P, I),
    time(I),
    place(X),
    person(S, P).

 :- 2 { boat(X, I) : place(X) }, time(I).
 :- { boat(X, I) : place(X) } 0, time(I).
 :- boat(X, I), boat(X, I+1), place(X), time(I).
 :- 3 { row(S, P, I) : person(S, P) }, time(I).
 :- { row(S, P, I) : person(S, P) } 0, time(I).

 :- at(X, woman, P_1, I),
    not at(X, man, P_1, I),
    at(X, man, P_2, I),
    pair(P_1 ; P_2), P_1 != P_2,
    place(X), time(I).
```

```
 :- row(man, P_1, I),
    not two_move(I),
    not has_company(woman, P_2, I),
    P_1 != P_2,
    pair(P_1 ; P_2),
    time(I).

two_move(I) :-
    2 { row(S, P, I) : person(S, P) },
    time(I).

has_company(woman, P_1, I) :-
    1 { at(X, S, P, I) : person(S, P)
                       : P != P_1 },
    at(X, woman, P_1, I),
    pair(P_1),
    place(X),
    time(I).

at(left, S, P, 1) :- person(S, P).
 :- 1 { not at(right, S, P, t+1)
            : person(S, P) } .
```

A.2 Heavily Optimized Planning Puzzle

```
pair(1..p).
time(1..t).
place(left; right; island).
sex(woman; man).
person(S, P) :- sex(S), pair(P).

{ row(S, P, I) } :-
    at(From, S, P, I),
    boat(From, I),
    time(I),
    place(From),
    person(S, P).

{ boat(L, I) : place(L) } :- time(I).
boat(right, t+1).

at(To, S, P, I+1) :-
    row(S, P, I),
    boat(To, I+1),
    time(I),
    place(To),
    person(S, P).

at(X, S, P, I+1) :-
```

```
    at(X, S, P, I),
    not row(S, P, I),
    time(I),
    place(X),
    person(S, P).

 :- 2 { boat(X, I) : place(X) }, time(I).
 :- { boat(X, I) : place(X) } 0, time(I).
 :- boat(X, I), boat(X, I+1), place(X), time(I).
 :- 3 { row(S, P, I) : person(S, P) }, time(I).
 :- { row(S, P, I) : person(S, P) } 0, time(I).

 :- at(X, woman, P_1, I),
    not at(X, man, P_1, I),
    at(X, man, P_2, I),
    pair(P_1 ; P_2), P_1 != P_2,
    place(X), time(I).

 :- row(man, P_1, I),
    not two_move(I),
    not has_company(woman, P_2, I),
    P_1 != P_2,
    pair(P_1 ; P_2),
    time(I).

two_move(I) :-
    2 { row(S, P, I) : person(S, P) },
    time(I).

has_company(woman, P_1, I) :-
    1 { at(X, S, P, I) : person(S, P)
                       : P != P_1 },
    at(X, woman, P_1, I),
    pair(P_1),
    place(X),
    time(I).

at(left, S, P, 1) :- person(S, P).

 :- 1 { not at(right, S, P, t+1)
            : person(S, P) }.

%% Optimizations
% No identical moves:
id_move(I) :-
    row(S, P_1, I),
    row(S, P_2, I),
    row(S, P_1, I+1),
    row(S, P_2, I+1),
```

```
    time(I), I < t,
    sex(S),
    pair(P_1; P_2), P_1 < P_2.

id_move(I) :-
    row(S_1, P, I),
    row(S_2, P, I),
    row(S_1, P, I+1),
    row(S_2, P, I+1),
    time(I), I < t,
    sex(S_1 ; S_2), S_1 < S_2,
    pair(P).

id_move(I) :-
    row(S, P, I),
    row(S, P, I+1),
    not two_move(I),
    not two_move(I+1),
    time(I), I < t,
    person(S, P).

 :- id_move(I), time(I), I < t.

 :- row(woman, P_1, I),
    row(man, P_2, I),
    pair(P_1 ; P_2), P_1 != P_2,
    time(I).

equivalent(P_1, P_2, I) :-
    at(L_1, S_1, P_1, I),
    at(L_1, S_1, P_2, I),
    at(L_2, S_2, P_1, I),
    at(L_2, S_2, P_2, I),
    pair(P_1 ; P_2), P_1 < P_2,
    sex(S_1 ; S_2), S_1 < S_2,
    place(L_1 ; L_2),
    time(I).

 :- equivalent(P_1, P_2, I),
    row(S, P_2, I),
    not row(S, P_1, I),
    pair(P_1 ; P_2), P_1 < P_2,
    sex(S),
    time(I).
```

A.3 Sokoban

```
time(1..n).
direction( north ; east ; west ; south).
```

```
% We only try to move to places from where there
% is a route to a target location:
move_square(X, Y) :-
    square(X, Y),
    has_target_route(X, Y).

has_target_route(X, Y) :-
    target_square(X, Y).

has_target_route(X, Y) :-
    square(X, Y ; X+1, Y; X-1, Y),
    has_target_route(X+1, Y).

has_target_route(X, Y) :-
    square(X, Y ; X-1, Y; X+1, Y),
    has_target_route(X-1, Y).

has_target_route(X, Y) :-
    square(X, Y ; X, Y+1; X, Y-1),
    has_target_route(X, Y+1).

has_target_route(X, Y) :-
    square(X, Y ; X, Y+1; X, Y-1),
    has_target_route(X, Y-1).

% A box may be pushed if it can be pushed:
{ push(X, Y, Dir, I) } :-
    can_push(X, Y, Dir, I),
    has_box(X, Y, I),
    has_neighbor(X, Y, Dir),
    possible_box(X, Y, I),
    time(I).

% No two boxs may be pushed at one time
 :- 2 { push(X, Y, Dir, I) : move_square(X, Y)
                           : direction(Dir) },
    time(I).

% A box ends where it is pushed to
has_box(X, Y, I+1) :-
    move_to(X, Y, I),
    move_square(X, Y),
    time(I).

% A box moves away when pushed
-has_box(X, Y, I+1) :-
    push(X, Y, Dir, I),
    time(I),
```

```
    has_neighbor(X, Y, Dir).

% A box stays at a place if not pushed
has_box(X, Y, I+1) :-
    not -has_box(X, Y, I+1),
    has_box(X, Y, I),
    time(I),
    move_square(X, Y).

% A box ends to exactly one position
% along the push direction
1 {move_to(X_2, Y_2, I)
   : same_segment(X_1, Y_1, X_2, Y_2, Dir)} 1 :-
    push(X_1, Y_1, Dir, I),
    has_neighbor(X_1, Y_1, Dir),
    time(I), I < n.

% A box may not be pushed over another
 :- has_box(X_2, Y_2, I),
    push(X_1, Y_1, Dir, I),
    move_to(X_3, Y_3, I),
    time(I),
    same_segment(X_1, Y_1, X_2, Y_2, Dir),
    same_segment(X_2, Y_2, X_3, Y_3, Dir).

% A box may not be pushed onto another
 :- has_box(X, Y, I),
    move_to(X, Y, I),
    move_square(X, Y),
    time(I).

% This predicate is used in optimizings
push_dir(Dir, I) :-
    push(X, Y, Dir, I),
    has_neighbor(X, Y, Dir),
    time(I).

% After pushing, sokoban is "at" the point
% where the moved box was
at(X, Y, I+1) :-
    push(X, Y, Dir, I),
    move_square(X, Y),
    has_neighbor(X, Y, Dir),
    time(I).

% A box can be pushed in a direction if the
% worker can reach it and there is space
% immediately in front of it.
```

```
can_push(X, Y, east, I) :-
    has_box(X, Y, I),
    move_square(X, Y ;  X+1, Y),
    square(X-1, Y),
    not has_box(X-1, Y, I),
    not has_box(X+1, Y, I),
    reachable(X-1, Y, I),
    possible_box(X, Y, I),
    time(I).
can_push(X, Y, west, I) :-
    has_box(X, Y, I),
    move_square(X, Y ; X-1, Y ),
    square(X+1, Y),
    possible_box(X, Y, I),
    not has_box(X-1, Y, I),
    not has_box(X+1, Y, I),
    reachable(X+1, Y, I),
    time(I).
can_push(X, Y, north, I) :-
    has_box(X, Y, I),
    square(X, Y-1),
    possible_box(X, Y, I),
    move_square(X, Y ; X, Y +1),
    not has_box(X, Y-1, I),
    not has_box(X, Y+1, I),
    reachable(X, Y-1, I),
    time(I).
can_push(X, Y, south, I) :-
    has_box(X, Y, I),
    move_square(X, Y-1 ; X, Y),
    not has_box(X, Y-1, I),
    not has_box(X, Y+1, I),
    square(X, Y +1),
    possible_box(X, Y, I),
    reachable(X, Y+1, I),
    time(I).

% The worker can reach all places that are not
% blocked.
reachable(X, Y, I) :-
    square(X, Y),
    time(I),
    at(X, Y, I).
reachable(X+1, Y, I) :-
    square(X, Y ; X+1, Y),
    time(I),
    reachable(X, Y, I),
    not has_box(X+1, Y, I).
reachable(X-1, Y, I) :-
```

```
    square(X, Y ; X-1, Y),
    time(I),
    reachable(X, Y, I),
    not has_box(X-1, Y, I).
reachable(X, Y+1, I) :-
    square(X, Y ; X, Y+1),
    time(I),
    reachable(X, Y, I),
    not has_box(X, Y+1, I).
reachable(X, Y-1, I) :-
    square(X, Y ; X, Y-1),
    time(I),
    reachable(X, Y, I),
    not has_box(X, Y-1, I).

% The initial situation:
at(X, Y, 1) :-
    initial_at(X, Y).
has_box(X, Y, 1) :-
    initial_box(X, Y).

% When squares have neighbors:
has_neighbor(X, Y, east) :-
    move_square(X, Y ; X +1, Y).
has_neighbor(X, Y, west) :-
    move_square(X, Y ; X -1, Y).
has_neighbor(X, Y, north) :-
    move_square(X, Y ; X, Y+1).
has_neighbor(X, Y, south) :-
    move_square(X, Y ; X, Y-1).

% Find out the segments of the level
same_segment(X, Y, X+1, Y, east) :-
    move_square(X, Y),
    move_square(X, Y).

same_segment(X, Y, X-1, Y, west) :-
    move_square(X, Y),
    move_square(X-1, Y).

same_segment(X, Y, X, Y+1, north) :-
    move_square(X, Y),
    move_square(X, Y+1).

same_segment(X, Y, X, Y-1, south) :-
    move_square(X, Y),
    move_square(X, Y-1).

same_segment(X_1, Y, X_2+1, Y, east) :-
```

```
    same_segment(X_1, Y, X_2, Y, east),
    move_square(X_1, Y; X_2, Y; X_2+1, Y).

same_segment(X_1, Y, X_2-1, Y, west) :-
    same_segment(X_1, Y, X_2, Y, west),
    move_square(X_1, Y; X_2, Y; X_2-1, Y).

same_segment(X, Y_1, X, Y_2+1, north) :-
    same_segment(X, Y_1, X, Y_2, north),
    move_square(X, Y_1; X, Y_2; X, Y_2+1).

same_segment(X, Y_1, X, Y_2-1, south) :-
    same_segment(X, Y_1, X, Y_2, south),
    move_square(X, Y_1; X, Y_2; X, Y_2-1).

%%% Constraints pruning out the search space:
% The final move has to be to a target square:
1 { move_to(X_2, Y_2, n)
    : same_segment(X_1, Y_1, X_2, Y_2, Dir) :
    target_square(X_2, Y_2) } 1 :-
    push(X_1, Y_1, Dir, n),
    has_neighbor(X_1, Y_1, Dir).

% A box may not be pushed twice to the same
% direction:
 :- push_dir(Dir, I),
    move_to(X, Y, I),
    push(X, Y, Dir, I +1),
    has_neighbor(X, Y, Dir),
    time(I).

% no immediate undoing of a move, if the worker
% could reach the other side of the box before
 :- push(X, Y, west, I),
    push_dir(east, I),
    move_to(X, Y, I+1),
    time(I), I < n,
    has_neighbor(X, Y, west),
    reachable(X-2, Y, I).

 :- push(X, Y, east, I),
    move_to(X, Y, I+1),
    push_dir(west, I),
    time(I), I < n,
    has_neighbor(X, Y, east),
    reachable(X+2, Y, I).

 :- push(X, Y, north, I),
    push_dir(south, I),
```

```
    move_to(X, Y, I+1),
    time(I), I < n,
    has_neighbor(X, Y, north),
    reachable(X, Y+2, I).

 :- push(X, Y, south, I),
    push_dir(north, I),
    move_to(X, Y, I+1),
    time(I), I < n,
    has_neighbor(X, Y, north),
    reachable(X, Y-2, I).

% Check if goal is reached:
goal(I) :- { not has_box(X, Y, I)
                 : target_square(X, Y) } 0,
    time(I).

% Do not push after goal is reached:
 :- 1 { move_to(X, Y, I) : move_square(X, Y) },
    time(I),
    goal(I).

% Treat first two moves special because we know
% what moves are possible:
possible_box(X, Y, 1) :-
    initial_box(X, Y).

possible_box(X_2, Y_2, 2) :-
    same_segment(X_1, Y_1, X_2, Y_2, Dir),
    move_square(X_2, Y_2),
    initial_box(X_1, Y_1).

possible_box(X, Y, 2) :-
    initial_box(X, Y).

possible_box(X, Y, I) :-
    time(I), I >= 3,
    move_square(X, Y).

% Don't push two boxes adjacent each other along
% the edge.
edge_pair(X, Y, X+1, Y) :-
    move_square(X, Y ; X+1, Y),
    not target_square(X, Y),
    not target_square(X+1, Y),
    not square(X, Y-1),
    not square(X+1, Y-1).

edge_pair(X, Y, X+1, Y) :-
```

```
    move_square(X, Y ; X+1, Y),
    not target_square(X, Y),
    not target_square(X+1, Y),
    not square(X, Y-1),
    not square(X+1, Y+1).

edge_pair(X, Y, X+1, Y) :-
    move_square(X, Y ; X+1, Y),
    not target_square(X, Y),
    not target_square(X+1, Y),
    not square(X+1, Y-1),
    not square(X, Y+1).

edge_pair(X, Y,  X+1, Y) :-
    move_square(X, Y ; X+1, Y),
    not target_square(X, Y),
    not target_square(X+1, Y),
    not square(X, Y+1),
    not square(X+1, Y+1).

edge_pair(X, Y, X, Y+1) :-
    move_square(X, Y ; X, Y+1),
    not target_square(X, Y),
    not target_square(X, Y+1),
    not square(X-1, Y),
    not square(X-1, Y+1).

edge_pair(X, Y, X, Y+1) :-
    move_square(X, Y ; X, Y+1),
    not target_square(X, Y),
    not target_square(X, Y+1),
    not square(X+1, Y),
    not square(X-1, Y+1).

edge_pair(X, Y, X, Y+1) :-
    move_square(X, Y ; X, Y+1),
    not target_square(X, Y),
    not target_square(X, Y+1),
    not square(X-1, Y),
    not square(X+1, Y+1).


edge_pair(X, Y, X, Y+1) :-
    move_square(X, Y ; X, Y+1),
    not target_square(X, Y),
    not target_square(X, Y+1),
    not square(X+1, Y),
    not square(X+1, Y+1).
```

```
 :- edge_pair(X_1, Y_1, X_2, Y_2),
    time(I),
    has_box(X_1, Y_1, I),
    has_box(X_2, Y_2, I).

% Don't push three boxes into a L-turn:
l_turn(X,Y, X+1, Y, X+1, Y+1) :-
    move_square(X, Y ; X+1, Y ; X+1, Y+1),
    not square(X+1, Y).

l_turn(X,Y, X+1, Y, X+1, Y-1) :-
    move_square(X, Y ; X+1, Y ; X+1, Y-1),
    not square(X, Y-1).

l_turn(X, Y, X, Y+1, X+1, Y+1) :-
    move_square(X, Y ; X, Y+1 ; X+1, Y+1),
    not square(X+1, Y).

l_turn(X, Y, X, Y+1, X+1, Y) :-
    move_square(X, Y ; X, Y+1 ; X+1, Y),
    not square(X+1, Y+1).

 :- { target_square(X_1, Y_1),
      target_square(X_2, Y_2),
      target_square(X_3, Y_3) } 2,
    l_turn(X_1, Y_1, X_2, Y_2, X_3, Y_3),
    time(I),
    has_box(X_1, Y_1,I),
    has_box(X_2, Y_2,I),
    has_box(X_3, Y_3,I).

% Don't form a 4-box square:
 :- move_square(X, Y),
    move_square(X+1,Y),
    move_square(X+1, Y+1),
    move_square(X, Y+1),
    time(I),
    not target_square(X, Y),
    not target_square(X+1,Y),
    not target_square(X+1, Y+1),
    not target_square(X, Y+1),
    has_box(X, Y, I),
    has_box(X+1, Y, I),
    has_box(X+1, Y+1, I),
    has_box(X, Y+1, I).

% In the end, all target squares must have boxes:
compute { has_box(X, Y, n +1)
             : target_square(X, Y) }.
```

```
hide .
show reachable(X, Y, I).
show push(X, Y, Dir, I).
show move_to(X, Y, I).
show has_box(X, Y, I).
show at(X, Y, I).
```

A.4 Finite Automata Constructor

```
#option -dn -Wall

const n = 4.      % number of NFA states
const min_d = 7.  % minimum number of DFA states
const max_d = 9.  % maximum number of DFA states
const min_f = 2.  % minimum number of accepting
                  %  DFA states
const max_f = 3.  % maximum number of accepting
                  %  DFA states
const min_E = 6.  % minimum number of edges in NFA
const max_E = 20. % maximum number of edges in NFA
const min_e = 2.  % minimum number of empty edges
                  %  in NFA
const max_e = 3.  % maximum number of empty edges
                  %  in NFA
const min_s = 3.  % The minimum number of
                  %  transitions per symbol
const min_a = 2.  % minimum number of states
                  % having conflicting transitions
const max_a = 4.  % maximum number of states
                  %  having conflicting transit-
                  %  ions

nd_state(1..n).
symbol(a ; b).
nd_symbol(X) :- symbol(X).
nd_symbol(_e_).

% NFA generation
{ nd_transition(Q_1, S, Q_2) } :-
    nd_state(Q_1 ; Q_2),
    nd_symbol(S).

{ nd_final(Q) : nd_state(Q) }.

ok_E :- min_E { nd_transition(Q_1, S, Q_2)
                : nd_state(Q_1 ; Q_2)
                : nd_symbol(S) } max_E.
```

```
ok_e :- min_e { nd_transition(Q_1, _e_, Q_2)
                : nd_state(Q_1 ;Q_2) } max_e.
ok_a :- min_a { has_two(Q) : nd_state(Q)} max_a.

has_two(Q_1) :-
    2 { nd_transition(Q_1, S, Q_2)
          : nd_state(Q_2) },
    nd_symbol(S),
    nd_state(Q_1).

% If only empty transitions leave from a state,
% there has to be at least two of them.
 :- { nd_transition(Q_1, _e_, Q_2)
        : nd_state(Q_2) } 1,
    { nd_transition(Q_1, S, Q_3) : nd_state(Q_3)
                                 : symbol(S) } 0,
    nd_state(Q_1).

 :- 1 { not ok_e, not ok_E, not ok_a }.

nd_reachable(1).
nd_reachable(Q_2) :-
    nd_reachable(Q_1),
    nd_transition(Q_1, S, Q_2),
    nd_state(Q_1; Q_2),
    Q_1 != Q_2,
    nd_symbol(S).

 :- nd_state(Q), not nd_reachable(Q).

nd_accepts(Q) :- nd_final(Q), nd_state(Q).
nd_accepts(Q_1) :-
    nd_accepts(Q_2),
    nd_transition(Q_1, S, Q_2),
    nd_state(Q_1; Q_2),
    Q_1 != Q_2,
    nd_symbol(S).

 :- nd_state(Q), not nd_accepts(Q).

 :- { nd_transition(Q_1, S, Q_2)
      : nd_state(Q_1; Q_2) } min_s - 1,
    symbol(S).

 :- nd_transition(Q, _e_, Q), nd_state(Q).

% Determinisation
in(4,15). in(4,12).   in(3,7).
In(3,15). in(3,12).   in(2,7).
```

```
in(2,15). in(4,11).   in(1,7).
in(1,15). in(2,11).   in(3,6).
in(4,14). in(1,11).   in(2,6).
in(3,14). in(4,10).   in(3,5).
in(2,14). in(2,10).   in(1,5).
in(4,13). in(4,9).    in(3,4).
in(3,13). in(1,9).    in(2,3).
in(1,13). in(4,8).    in(1,3).
in(2,2).  in(1,1).
d_state(0 .. 15).

closure(Q, Q) :- nd_state(Q).
closure(Q_1, Q_2) :-
    nd_state(Q_1 ; Q_2),
    Q_1 != Q_2,
    nd_transition(Q_1, _e_, Q_2).

closure(Q_1, Q_3) :-
    nd_transition(Q_2, _e_, Q_3),
    closure(Q_1, Q_2), Q_1 != Q_2,
    Q_2 != Q_3, Q_1 != Q_3,
    nd_state(Q_1 ; Q_2 ; Q_3).

impossible(Q) :-
    d_state(Q),
    in(N_1, Q),
    closure(N_1, N_2),
    not in(N_2, Q),
    nd_state(N_2), N_2 != N_1.

d_initial(Q) :-
    d_state(Q),
    in(1, Q),
    not impossible(Q),
    not not_initial(Q).

not_initial(Q) :-
    in(N, Q),
    not closure(1, N).

d_transition(Q_1, S, Q_2) :-
    d_reachable(Q_1),
    not impossible(Q_2),
    not wrong(Q_1, S, Q_2),
    d_state(Q_1; Q_2),
    symbol(S).

wrong(Q_1, S, Q_2) :-
    in(N_1, Q_1),
```

```
    not in(N_2, Q_2),
    d_state(Q_2),
    nd_state(N_2),
    nd_transition(N_1, S, N_2),
    symbol(S).

wrong(Q_1, S, Q_2) :-
    in(N, Q_2),
    not has_transition(Q_1, S, N),
    d_state(Q_1), symbol(S).

has_transition(Q_1, S, N_2) :-
    in(N_1, Q_1),
    nd_transition(N_1, S, N_2),
    nd_state(N_2),
    symbol(S).

has_transition(Q_1, S, N_2) :-
    has_transition(Q_1, S, N_1),
    closure(N_1, N_2),
    d_state(Q_1), symbol(S), nd_state(N_1; N_2).

d_reachable(Q) :- d_initial(Q), d_state(Q).
d_reachable(Q_2) :-
    d_transition(Q_1, S, Q_2),
    d_reachable(Q_1),
    d_state(Q_1 ; Q_2), Q_1 != Q_2,
    symbol(S).

d_final(Q) :-
    d_reachable(Q),
    in(N, Q),
    nd_final(N).

ok_d :- min_d { d_reachable(Q) : d_state(Q) } max_d.
ok_f :- min_f { d_final(Q) : d_state(Q) } max_f.
 :- 1 { not ok_d, not ok_f }.

compute 0 { }.
```

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A73 Toni Jussila
Bounded Model Checking for Verifying Concurrent Programs. August 2002.

HUT-TCS-A74 Sam Sandqvist
Aspects of Modelling and Simulation of Genetic Algorithms: A Formal Approach.
September 2002.

HUT-TCS-A75 Tommi Junttila
New Canonical Representative Marking Algorithms for Place/Transition-Nets. October 2002.

HUT-TCS-A76 Timo Latvala
On Model Checking Safety Properties. December 2002.

HUT-TCS-A77 Satu Virtanen
Properties of Nonuniform Random Graph Models. May 2003.

HUT-TCS-A78 Petteri Kaski
A Census of Steiner Triple Systems and Some Related Combinatorial Objects. June 2003.

HUT-TCS-A79 Heikki Tauriainen
Nested Emptiness Search for Generalized Büchi Automata. July 2003.

HUT-TCS-A80 Tommi Junttila
On the Symmetry Reduction Method for Petri Nets and Similar Formalisms.
September 2003.

HUT-TCS-A81 Marko Mäkelä
Efficient Computer-Aided Verification of Parallel and Distributed Software Systems.
November 2003.

HUT-TCS-A82 Tomi Janhunen
Translatability and Intranslatability Results for Certain Classes of Logic Programs.
November 2003.

HUT-TCS-A83 Heikki Tauriainen
On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata.
December 2003.

HUT-TCS-A84 Johan Wallén
On the Differential and Linear Properties of Addition. December 2003.

HUT-TCS-A85 Emilia Oikarinen
Testing the Equivalence of Disjunctive Logic Programs. December 2003.

HUT-TCS-A86 Tommi Syrjänen
Logic Programming with Cardinality Constraints. December 2003.

ISBN 951-22-6896-5
ISSN 1457-7615