Cardinality Constraint Programs

Tommi Syrjänen*

Helsinki University of Technology, Dept. of Computer Science and Eng.,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FIN-02015 HUT, Finland
Tommi.Syrjanen@hut.fi

Abstract. We define the class of cardinality constraint logic programs and provide a formal stable model semantics for them. The class extends normal logic programs by allowing the use of cardinality constraints and conditional literals. We identify a decidable subset, omega-restricted programs, of the class. We show how the formal semantics can be extended to allow the use of evaluated function symbols, such as arithmetic built-in operators. The omega-restricted cardinality constraint programs have been implemented in the Smodels system.

1 Introduction

When we use Answer Set Programming (ASP) to solve a problem, we encode it using some logical framework so that the solutions correspond with the models of the system, and then use a solver of the chosen formalism to find them. What differentiates ASP from the traditional logic programming is that an ASP solution is a set of literals instead of a proof tree of Prolog and its derivatives.

Most ASP systems use the stable model semantics [6] of logic programs as their underlying semantics ([12, 4, 1]) but systems based on propositional logic ([5]) also exist. Most logic program -based systems have some extensions to the basic inference rules. For example, `dlv` [4] allows disjunctive rules and SMODELS [12] has cardinality and weight constraint literals.

The usual way to define a semantics for ASP is to start with variable-free (ground) programs and then say that the variables are just short-hand constructs for denoting sets of rules. This approach has the advantage that you do not have to alter the basic semantics to use variables and in most cases it is easy to see what set of ground rules correspond to a rule with variables. However, the things get more complex when we allow the use of function symbols, especially when some of function symbols are built-ins that should be evaluated (like $+$, $-$) and some are uninterpreted Herbrand terms.

In this work we examine in detail how formal stable model semantics can be defined for non-ground logic programs with cardinality constraints and conditional literals, and explain the motivation why the definitions are done the way they are. We do it in four steps: 1) we define the semantics for ground programs;

* This work has been supported by the Academy of Finland (project 53695).

2) add variables to them; 3) define some syntactic sugar to make the language easier to use; and 4) identify a decidable subset that corresponds to the language used in the SMODELS system.[1] Finally, we show how interpreted built-in functions fit into the formal semantics. This work is based on [11] and extends [10]. Non-ground cardinality and weight constraints have been examined also earlier ([9, 8]). This work has two major differences from the precious work: (1) we allow the default negation of cardinality constraint literals; and (2) the treatment of conditional literals is now more precise.

In the language definition we identify basic primitives and the more complex constructs are then translated into programs that use only those primitives. The main criteria that we use in selecting the primitives are:

1. A program with variables can always be replaced by a ground program that has the same set stable models;
2. When an extended program is translated into basic primitives no new atoms should be generated;
3. It should be possible to translate a complex program with variables without having to instantiate the program first; and
4. All translations should be linear in size.

The first criterion means that we keep the standard interpretation of variables as short-hand notation for sets of ground rules as this makes defining the semantics more convenient. If function symbols are used, then the corresponding ground program is infinite. The second criterion is included since we want to keep a very close link between the original extended program and the translated simple one. It also allows us to manipulate the programs by combining or splitting them without having to worry about possible clashes introduced by the new atoms. The third criterion is connected to the first one with the idea being that we can always see the intended meaning of the program without having to create the potentially infinite instantiation. A translation is linear if the number of new literals and rules is some constant times the size of the original construct.

2 Language

A *term* is either a *variable* or a *function term* $f(t_1, \ldots, t_k)$ where f is a k-ary function symbol and t_1, …, t_k are terms. A 0-ary function symbol is a *constant.* A term is *ground* if it does not contain any variables.

An *atom* is of the form $p(t_1, \ldots, t_k)$ where p is a k-ary predicate symbol and t_1, …, t_k are terms. An atom is ground iff all terms in it are ground. In the following we use A to denote an atom if we are not interested in its arguments, and $\mathrm{pred}(A)$ to denote its predicate symbol. The symbol $\top$ denotes a special atom that is always true. A *basic literal* is either an atom A (positive) or its negation not A (negative). We use L to denote a basic literal and $\overline{L}$ its complement when their arguments are not relevant. A *conditional literal* L_{c} is of the form:

$$X.L : A \tag{1}$$

[1] However, the current version of SMODELS recognizes a slightly simpler language.

where the *main literal* L is a basic literal, the *condition* A is an atom, and X is a set of *local variables.* If $X = \emptyset$, we denote (1) simply as $L : A$, and if X is a singleton set we write it without braces. All variables that occur in L_{c} that are not local are *global.* Intuitively $L : A$ can be seen as a conjunction that is evaluated in two phases: first A is checked, and if it is true, then the truth value of L determines the truth value of the whole construct. A conditional literal is positive if L is, and it is negative otherwise. A *literal* is either a basic literal or a conditional literal. We use the notation $\mathcal{L}$ to denote literals.

A *cardinality constraint* C is of the form:

$$C = \mathrm{Card}(b, S) \tag{2}$$

where b is an integral *bound* and S is a set of literals. The basic intuition is that C is true if the number of true literals $\mathcal{L} \in S$ is greater than or equal to b.[2] The set of positive literals in S is denoted by $\mathrm{pos}(C)$ and the set of negative literals by $\mathrm{neg}(C)$. A *cardinality constraint literal* $\mathcal{C}$ is either a cardinality constraint C or its negation not C.

A *basic rule* is of the form:

$$A \leftarrow \mathcal{C}_1, \ldots, \mathcal{C}_n \tag{3}$$

where the *head* A is an atom, and $\mathcal{C}_i$ in its *body* are cardinality constraint literals. The rule (3) encodes the fact that if all literals in the body are true, then the head must also be true. If the body is empty ($n = 0$), then we call a basic rule a *fact.* As above, $\mathrm{pos}(R)$ and $\mathrm{neg}(R)$ denote the sets of positive and negative cardinality constraint literals in the rule body.

We immediately define a two shorthand notations that are used in our examples: 1) a basic literal L in a rule body denotes the constraint literal $\mathrm{Card}(1, \{L\})$; and 2) an empty head is replaced by a new atom f that does not occur anywhere in the program and a rule $f' \leftarrow f, \mathrm{not}\ f'$ that causes a contradiction if f is true to the program. Thus, such a rule acts as a constraint on the models and a model candidate that makes its body true is rejected.

A *choice rule* has the form:

$$\{A\} \leftarrow \mathcal{C}_1, \ldots, \mathcal{C}_n \tag{4}$$

where A and $\mathcal{C}_i$ are defined as above. The intuition is that if the rule body is true, then A may be true but it does not have to be true. A cardinality constraint program is a set of rules.

For each formula (term, literal, cardinality constraint, or rule) F, $\mathrm{Var}(F)$ denotes the set of variables that occur in it.

These definitions conclude our basic language. In Section 4 we define several extended language constructs that are then translated to the basic language.

[2] In SMODELS syntax this written as $b\ S$

3 Stable Model Semantics

3.1 Ground Programs

In this section we define the stable model semantics for the basic cardinality constraint programs. We do the definition in stages, starting from the simplest possible programs and then extend the definitions to cover the full basic language. In the definition we will be using the notation $M \models F$ to denote that a set of ground atoms M satisfies the formula F. In particular, for an atom A, $M \models A$ iff $A \in M$ and $M \models \text{not } A$ iff $A \notin M$. In case of a cardinality constraint $\text{Card}(b, S)$ where S contains only basic literals, $M \models \text{Card}(b, S)$ iff $b \leq |\{L \in S \mid M \models L\}|$, and $M \models \text{not Card}(b, S)$ iff $M \not\models \text{Card}(b, S)$.

In the simplest case the rules have only basic literals and positive cardinality constraint literals in their bodies. We call these *simple* rules. They are essentially equivalent to the extended programs presented in [8].

Definition 1. *Let $C = \text{Card}(b, S)$ be a ground cardinality constraint and M be a set of ground atoms. Then, the* reduct *C^M is the cardinality constraint:*

$$C^M = \text{Card}(b', \text{pos}(C)) \tag{5}$$

where $b' = b - |\{L \in \text{neg}(C) \mid M \models L\}|$. The reduct R^M of a ground simple basic rule $R = A \leftarrow C_1, \ldots, C_n$ is the singleton set of rules:

$$R^M = \{A \leftarrow C_1^M, \ldots, C_n^M\} \tag{6}$$

and the reduct of a ground simple choice rule $R = \{A\} \leftarrow C_1, \ldots, C_n$ is the set of rules:

$$R^M = \begin{cases} \{A \leftarrow C_1^M, \ldots, C_n^M\}, & \textit{if } A \in M \\ \emptyset, & \textit{otherwise} \ . \end{cases} \tag{7}$$

A reduct of a set P of ground simple rules is the set:

$$P^M = \bigcup_{R \in P} R^M \ . \tag{8}$$

Example 1. Consider the rule $R = \{a\} \leftarrow \text{Card}(2, \{b, \text{not } c\})$ and let $M = \{a\}$. Then, $R^M = \{a \leftarrow \text{Card}(1, \{b\})\}$.

Note that all rules that belong to a reduct of a program P are basic rules and all basic literals that occur in them are positive. Such rules are monotonous [8] so the reduct P^M has a unique least model that we denote with $\mathbf{MM}(P^M)$. The least model is the least fixpoint of the operator T_P where $T_P(S) = \{A \mid A \leftarrow C_1^M, \ldots, C_n^M \in P^M \text{ and } S \models C_1^M, \ldots, C_n^M\}$ [8]. If this least model happens to coincide with M, then M is a stable model of P.

Definition 2. *Let P be a ground simple cardinality constraint program. A set of ground atoms M is a* stable model *of P if and only if:*

$$\mathbf{MM}(P^M) = M \ . \tag{9}$$

Next, we extend the semantics by allowing ground conditional literals as well as negative cardinality constraints. We add an extra step, *expansion*, that is done before reduction and at this point all conditional literals $L : A$ are either replaced by L if A is true or removed altogether if A is false.

Definition 3. *Let M be a ground set of atoms. Then, the* expansion *of a ground basic literal L with respect to M is* $\mathrm{E}(L, M) = \{L\}$*, and the expansion of a ground conditional literal $L_\mathrm{c} = L : A$ is*

$$\mathrm{E}(L_\mathrm{c}, M) = \begin{cases} \{L\}, & A \in M \\ \emptyset, & \textit{otherwise} \end{cases} . \tag{10}$$

The expansion of a ground cardinality constraint $\mathrm{Card}(b, S)$ *with respect to M is the cardinality constraint:*

$$\mathrm{Card}(b, \bigcup_{\mathcal{L} \in S} \mathrm{E}(\mathcal{L}, M)) . \tag{11}$$

Example 2. Let $C = \mathrm{Card}(1, \{a : \top, \mathrm{not}\ b, c : d, e : f\})$ and $M = \{d\}$. Then, $E(C, M) = \mathrm{Card}(1, \{a, \mathrm{not}\ b, c\})$.

We obtain the reduct of a ground rule by first expanding all constraint literals in its body, and then removing the negative constraints from its body in a way that is analogous to the original Gelfond-Lifschitz -reduction [6].

Definition 4. *Let $R = A \leftarrow C_1, \ldots, C_n$, $\mathrm{not}\ C'_1, \ldots, \mathrm{not}\ C'_m$ be a basic cardinality constraint rule and M a set of ground atoms. Then, the* reduct R^M *is:*

$$R^M = \begin{cases} \{A \leftarrow \mathrm{E}(C_1, M)^M, \ldots, \mathrm{E}(C_n, M)^M\}, & M \models \mathrm{E}(C'_1, M)^M, \ldots, \\ & \qquad \mathrm{E}(C'_m, M)^M \\ \emptyset, & \textit{otherwise} \end{cases} . \tag{12}$$

Let $R = \{A\} \leftarrow C_1, \ldots, C_n$, $\mathrm{not}\ C'_1, \ldots, \mathrm{not}\ C'_m$ be a choice rule. Then,

$$R^M = \begin{cases} \{A \leftarrow \mathrm{E}(C_1, M)^M, \ldots, \mathrm{E}(C_n, M)^M\}, & M \models \mathrm{E}(C'_1, M)^M, \ldots, \\ & \qquad \mathrm{E}(C'_m, M)^M \\ & \textit{and } A \in M \\ \emptyset, & \textit{otherwise} \end{cases} . \tag{13}$$

As was the case with the simple programs, M is a stable model of P if and only if $P = \mathbf{MM}(P^M)$.

Example 3. Let P be the program:

$$\{a\} \leftarrow \mathrm{not}\ \mathrm{Card}(1, \{b : a\})$$
$$\{b\} \leftarrow \mathrm{Card}(1, \{\mathrm{not}\ a\})$$

Now P has three stable models: $M_1 = \emptyset$, $M_2 = \{a\}$, and $M_3 = \{b\}$. In the case of M_2, the reduct $P^M = \{a \leftarrow\}$ since $M_2 \models \mathrm{not}\ \mathrm{Card}(1, \{b : a\})^M = \mathrm{not}\ \mathrm{Card}(1, \{b\})$.

3.2 Programs with Variables

A rule with variables denotes the set of ground rules that can be obtained by replacing each variable by terms of the Herbrand universe of the program. As there are two types of variables, local and global, the instantiation is defined in two parts. First, the local variables in conditional literals are replaced by their instantiations, and then the same is done for the global variables.

The *Herbrand universe* $\mathbf{HU}(P)$ of a cardinality constraint program P is the set of all ground terms that can be constructed using the function terms that occur in P.

Definition 5. *A* substitution *is a function* $\sigma_{V,U} : V \to U$ *that maps a set of variables* V *to an universe* U*. The set of all substitutions from* V *to* U *is denoted by* $\mathrm{Sub}(V, U)$*.*

A substitution *applied* to a variable v is the term:

$$v\sigma_{V,U} = \begin{cases} \sigma_{V,U}(v), & v \in V \\ v, & \text{otherwise,} \end{cases} \tag{14}$$

and a substitution applied to a function term $t = f(t_1, \ldots, t_n)$ is the term $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$. For atoms and literals the substitution is applied similarly to the case of function terms, that is, their arguments are substituted.

Definition 6. *Let* P *be a program and* $L_{\mathrm{c}} = X.L : A$ *be a conditional literal that occurs in it. Then, the* local instantiation $I(L_{\mathrm{c}}, P)$ *of* L_{c} *is the set:*

$$I(L_{\mathrm{c}}, P) = \{L\sigma : A\sigma \mid \sigma \in \mathrm{Sub}(X, \mathbf{HU}(P))\} \ . \tag{15}$$

The local instantiation of a basic literal L *is the set* $I(L, P) = \{L\}$*, and the local instantiation of a cardinality constraint* $C = \mathrm{Card}(b, S)$ *is the constraint* $I(C, P) = \mathrm{Card}(b, \bigcup_{\mathcal{L} \in P} I(\mathcal{L}, P))$*.*

Definition 7. *The* instantiation *of a rule* $R = A \leftarrow C_1, \ldots, C_n$ *is the set of rules:*

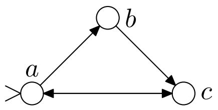
$$\mathbf{HI}(R, P) = \{A\sigma \leftarrow I(C_1, P)\sigma, \ldots, I(C_n, P)\sigma \mid \sigma \in \mathrm{Sub}(\mathrm{Var}(P), \mathbf{HU}(P))\} \tag{16}$$

The Herbrand instantiation *of a program* P *is the set of rules:*

$$\mathbf{HI}(P) = \bigcup_{R \in P} \mathbf{HI}(R, P) \ . \tag{17}$$

Example 4. Consider the following encoding H of the Hamilton cycle problem:

$$\begin{aligned} \{hc(X,Y)\} &\leftarrow arc(X,Y) \\ &\leftarrow \mathrm{Card}(2, \{Y.hc(X,Y) : arc(X,Y)\}), vtx(X) \\ r(Y) &\leftarrow \mathrm{Card}(1, \{start(X), r(X)\}), hc(X,Y), arc(X,Y) \\ &\leftarrow vtx(X), \mathrm{not}\ r(X). \end{aligned}$$



$$\begin{array}{rl} vtx(a). & arc(a,b). \\ vtx(b). & arc(a,c). \\ vtx(c). & arc(b,c). \\ start(a). & arc(c,a). \end{array}$$

Fig. 1. Sample graph for the Hamilton cycle problem

The first rule asserts that each arc of the graph may belong in the Hamiltonian cycle, and the second one ensures that at most one arc may leave from any vertex of the graph. The third rule computes the the set of visited vertices starting from an initial vertex, and the last rule ensures that all vertices belong to the cycle.

Next we add to the program the datacorresponding to the graph that is presented in Figure 1. Then, the local instantiations of the constraint literal in the second rule is: $\{hc(X,a) : arc(X,a), hc(X,b) : arc(X,b), hc(X,b) : arc(X,b)\}$, so the instantiation of the rule is:

$$\leftarrow \mathrm{Card}(2, \{hc(a,a) : arc(a,a), hc(a,b) : arc(a,b), hc(a,c) : arc(a,c)\}), vtx(a)$$
$$\leftarrow \mathrm{Card}(2, \{hc(b,a) : arc(b,a), hc(b,b) : arc(b,b), hc(b,c) : arc(b,c)\}), vtx(b)$$
$$\leftarrow \mathrm{Card}(2, \{hc(c,a) : arc(c,a), hc(c,b) : arc(c,b), hc(c,c) : arc(c,c)\}), vtx(c) \ .$$

4 The Extended Language

In this section we define several additional syntactic constructs for cardinality constraint programs. They are seen as notational shortcuts for larger basic language constructs. We have found that these extensions are useful in practice for modeling several different problem domains.

Upper Bounds for Constraints We allow a cardinality constraint to have also an upper bound u. The intuition is that a constraint $C_{\mathrm{u}} = \mathrm{Card}(b, u, S)$ is true iff the number of satisfied literals in it is between the bounds, inclusive. We can express C_{u} in basic syntax by replacing it with two constraint literals: $\mathrm{Card}(b, S)$ and not $\mathrm{Card}(u+1, S)$. The intuition is that C_{u} is true if the lower bound is met but it is not the case that the number of satisfied literals is strictly greater than the upper bound.

Conditional Literals in Rule Bodies We need a more complex translation if we want to use a conditional literal $X.L : A$. An intuitive semantics for such literal is universal quantification; the non-ground conditional literal should be true if $L\sigma$ is true when $A\sigma$ is. Thus, we replace it by a negative constraint literal:

$$\text{not } \mathrm{Card}(1, \{\overline{L} : A\}) \ . \tag{18}$$

This construct is analogous to the classical equality of $\forall x.p(x)$ and $\neg\exists x \neg p(x)$. However, in Section 5 we see that it differs from the classical case in one important sense.

Positive Cardinality Constraint Literals in Rule Heads Thus far we have had only atoms in rule heads. However, many problems have natural representations where there are cardinality constraints in the heads so we want to allow them with the restriction that all literals that occur in the head are positive. A rule:

$$\mathrm{Card}(b, u, \{X_1.A_1 : D_1, \ldots, X_n.A_n : D_n\}) \leftarrow \mathit{body} \tag{19}$$

is translated into $n + 2$ rules:

$$\begin{aligned} \{A'_i\} &\leftarrow D'_i, \mathit{body} \\ &\leftarrow \text{not } \mathrm{Card}(b, \{X_1.A_1 : D_1, \ldots, X_n.A_n : D_n\}), \mathit{body} \\ &\leftarrow \mathrm{Card}(u+1, \{X_1.A_1 : D_1, \ldots, X_n.A_n : D_n\}), \mathit{body} \end{aligned} \tag{20}$$

where A'_i and D'_i are obtained by renaming all local variables that occur in them. The first rule allows us to include any atom A_i in the model when the *body* is true while the next two rules ensure that the number of such atoms is between the bounds. In the first rule we effectively change local variables into global ones but this does not cause any problems since they stay local variables in the other two rules. If there is no upper bound, then the third rule is not necessary. Note also that if $b = 0$, then the second rule is trivially satisfied, so we obtain in that case behavior that is identical with choice rules.

Example 5. The rule $\mathrm{Card}(1, 2, \{X.a(X) : b(X)\}) \leftarrow c(X)$ is translated to

$$\begin{aligned} \{a(Y)\} &\leftarrow b(Y), c(X) \\ &\leftarrow \text{not } \mathrm{Card}(1, \{X.a(X) : b(X)\}), c(X) \\ &\leftarrow \mathrm{Card}(3, \{X.a(X) : b(X)\}), c(X) \enspace . \end{aligned}$$

5 Discussion on the Basic and Extended Languages

In this section we examine the motivations for the definitions in the previous two sections. The very first question is: why cardinality literals and no literals for other aggregate types? The immediate reason is that they form a simple and well-understood base so the semantics stays relatively clear and intuitive. This simple case can then be extended to more complex aggregates when necessary.

Another, not as obvious reason is that cardinality constraints can be used everywhere where a basic literal can occur in normal programs, including in rule heads in the case of the full language. This means that we can use the full power of non-monotonic reasoning with them. A common approach for defining the stable model semantics for aggregates is to use them like negative literals in the reduct: if an aggregate is satisfied by the model candidate, it is removed, but if it is unsatisfied, the whole rule is discarded [7]. However, this makes it possible for an aggregate to justify itself in the model even when all basic literals in it are positive and ground. For example, if this approach is taken in the program:

$$a \leftarrow \mathrm{Card}(1, \{a\}) \enspace ,$$

then $M = \{a\}$ is a valid stable model as $M \models \mathrm{Card}(1, \{a\})$ even though the only justification for a is a itself.

Note that our semantics allows a circular justification in case where an atom depends on itself via double negation, as in

$$a \leftarrow \mathrm{not}\ \mathrm{Card}(1, \{\mathrm{not}\ a\}) \ .$$

This example shows that $\mathrm{Card}(1, \{a\})$ and not $\mathrm{Card}(1, \{\mathrm{not}\ a\})$ are not equivalent even though they are indistinguishable in the classical sense. This situation is analogous to the normal programs where $a \leftarrow a$ and $a \leftarrow \mathrm{not}\ noa$; $noa \leftarrow \mathrm{not}\ a$ have different stable models.

Positive cycles are also the reason why the basic language does not have conditional literals in rule heads. Consider the hypothetical rule $R = \{a : a\} \leftarrow$ that encodes the silly condition that a may be true if a is true. Now, if we expanded the condition before taking the reduct, then $\{a\}$ would be a model of R even though the only justification for a is that a is true.

The combination of local variables and conditional literals is interesting in the sense that you can use them to implement existential quantification in rule bodies. In effect, a cardinality constraint $\mathrm{Card}(1, \{X.a(X) : b(X)\})$ encodes an existential quantification $\exists x.(a(x) \wedge b(x))$. A straightforward way to encode such a quantification using only normal programs, would need a new atom to do it. Similarly, we get an universal quantification $\forall x.(b(x) \rightarrow a(x))$ using not $\mathrm{Card}(1, \{X.\mathrm{not}\ a(X) : b(X)\})$. Encoding the universal quantification without conditional literals is tricky as you have to separate the cases where $b(X)$ is never true, so the implication is trivially true, from the cases where $b(X)$ is sometimes true.

6 Omega-Restricted Programs

If a logic program has function symbols in it, then its Herbrand instantiation is infinite and deciding whether it has a stable model or not becomes undecidable even when only Horn rules are used [3]. Also, even if the instantiation is finite, it may be exponential in size so constructing it may be intractable. Fortunately, in most cases most of the rules of the instantiation have trivially unsatisfiable bodies so we may leave them out without affecting the set of stable models.

Next we identify a decidable subset of cardinality constraint programs, namely *ω-restricted* programs [10]. The basic idea is to enforce syntactic restrictions on the programs so that it can be guaranteed that their all stable models are finite. The predicate symbols of a program are arranged into a stratification where more complex predicates are defined in terms of simpler ones. The stratification extends the usual definition of stratification [2] by adding a new level, the ω-stratum, to contain the unstratifiable part of the program.

A constraint literal is *simple* if it is of the form $\mathrm{Card}(1, \{A : \top\})$. 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. This condition ensures that stable models stay finite.

6.1 Dependency Graphs

We start by defining a few helper notations. The set $\text{bbody}^+(R)$ contains all positive basic literals and main literals of positive conditional literals that occur within positive cardinality constraint literals of the body of R. Thus,

$$\text{bbody}^+(R) = \bigcup_{C \in \text{pos}(R)} \{L \mid L \in \text{pos}(C) \text{ or } X.L : A \in \text{pos}(C)\} \ ,$$

while $\text{bbody}^-(R)$ contains all other basic and main literals. Finally, $\text{simple}(R)$ denotes the set of simple constraint literals in the body of R.

Definition 8. *A* dependency graph *of a program P is a triple $D_P = \langle V, E^+, E^- \rangle$ where V is the set of predicate symbols occurring in P, and $E^+, E^- \subseteq V \times V$ are the sets of* positive *and* negative *dependency edges, where $\langle p, q \rangle \in E^+$ iff there exists a rule $R \in P$ where p occurs in the head and q occurs in* $\text{bbody}^+(R)$*, and $\langle p, q \rangle \in E^-$ if at least one of the following conditions hold:*

1. *p occurs in a head of a rule $R \in P$ and $q \in \text{bbody}^-(R)$;*
2. *there is a conditional literal in P where p occurs as the main literal and q as the condition; or*
3. *$p = q$ and p occurs in the head of a choice rule.*

A dependency path *of a program P is a sequence $\langle p_1, \ldots, p_n \rangle$ of predicate symbols such that $\langle p_i, p_{i+1} \rangle \in E^+ \cup E^-$ for all $1 \leq i < n$. A path is* negative *if for some i, $\langle p_i, p_{i+1} \rangle \in E^-$.*

Intuitively, a predicate p depends on q if we cannot know the value of p without knowing the value of q. The definition of E^+ is straightforward, but the one for E^- is more involved. The first case of its definition correspond directly with the E^+ case. The second one is used to ensure that each time when we instantiate a conditional literal $X.L : A$, we already know the extension of A so we can leave out those instances of L that have unsatisfied conditions. The last case is an artificial definition that prevents us from mistakingly concluding that p has a fixed extension if one of its rules is a choice rule.

Example 6. The dependency graph of the Hamilton cycle program is shown in Figure 2. The solid lines are positive arcs and the dashed lines are negative.

Definition 9. *A predicate p* depends *on a predicate q (denoted $q \prec p$) in a program P iff there exists a dependency path $\langle p, \ldots, q \rangle$ in D_P. The dependence is* negative *(denoted $q \prec_- p$) iff there exists a negative dependency path $\langle p, \ldots, q \rangle$ in D_P.*

Next, we define the ω-stratification. A predicate has to be on at least as high stratum as the predicates that it depends on positively, and it has to be on a higher stratum if the dependency is negative unless both predicates are on the ω-stratum.

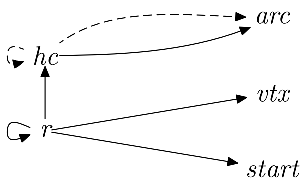


Fig. 2. The Dependency graph of the Hamilton cycle program

Definition 10. *An* ω-stratification*of a program P is a function $\mathcal{S} : \mathcal{P}(P) \to \mathbb{N} \cup \{\omega\}$ such that:*

1. $\forall p, q \in \mathcal{P}(P)$*: if $q \prec p$, then $\mathcal{S}(p) \geq \mathcal{S}(q)$;*
2. $\forall p, q, \in \mathcal{P}(p)$*: if $q \prec_{-} p$, then $\mathcal{S}(p) > \mathcal{S}(q)$ or $\mathcal{S}(p) = \mathcal{S}(q) = \omega$.*

An ω-stratification is strict *iff:*

1. *if $q \prec p$, $p \not\prec q$ and $\mathcal{S}(p) < \omega$, then $\mathcal{S}(p) > \mathcal{S}(q)$; and*
2. *if $\mathcal{S}(p) = \omega$, then there exists $q \in \mathcal{P}(P)$ such that $\mathcal{S}(q) = \omega$ and $q \prec_{-} p$.*

In [11] it is proved that every cardinality constraint program has a strict stratification and that all strict stratifications of a program are essentially equivalent.

Example 7. The Hamilton cycle program has the following strict ω-stratification: $\mathcal{S}(vtx) = \mathcal{S}(arc) = \mathcal{S}(start) = 0$, $\mathcal{S}(r) = \mathcal{S}(hc) = \omega$.

Definition 11. *The* ω-valuation *Ω of a rule R under an ω-stratification $\mathcal{S}$ is $\Omega(R, \mathcal{S}) = \mathcal{S}(pred(head(R)))$ and the valuation of a global variable V in a rule R is:*

$$\Omega(V, R, \mathcal{S}) = \min(\{\mathcal{S}(pred(A)) \mid A : \top \in \mathrm{simple}(R)\} \cup \{\omega\}) \tag{21}$$

Definition 12. *A conditional literal $X.L : A$ is* ω-restricted *under a stratification $\mathcal{S}$ iff $\mathcal{S}(L) > \mathcal{S}(A)$ and $X \subseteq \mathrm{Var}(A)$. A rule R is ω-restricted if all conditional literals in it are ω-restricted, and for all $V \in \mathrm{Var}(R)$, $\Omega(V, R, \mathcal{S}) < \Omega(R, \mathcal{S})$. A program P is ω-restricted if there exists a strict ω-stratification $\mathcal{S}$ such that all rules in P are ω-restricted under $\mathcal{S}$.*

Example 8. The rule $\{hc(X, Y)\} \leftarrow arc(X, Y)$ is ω-restricted under the stratification defined in Example 7 since $\Omega(R, \mathcal{S}) = \omega > \Omega(X, R, \mathcal{S}) = \Omega(Y, R, \mathcal{S}) = 0$.

Example 9. The rule $a(f(X)) \leftarrow a(X)$ is not ω-restricted since for each stratification $\mathcal{S}$, $\Omega(R, \mathcal{S}) = \Omega(X, R, \mathcal{S})$.

If a predicate belongs to a finite stratum in one strict stratification, then it does so in all such stratifications [11]. We call those predicates *domain predicates*.

Theorem 1. *The existence of a stable model of a finite ω-restricted cardinality constraint program is decidable.*

Proof. (Sketch, details are shown in [11]) We can construct a strict ω-stratification of a program P by finding the strongly connected components of its dependency graph [11]. After that, we can show by induction that at each stratum starting from the first one the stable model induced by the rules on that stratum is finite. At the first stratum all rules are ground so the models are trivially finite. In the following strata each variable that occurs in a rule occurs in a simple literal that belongs to an earlier stratum so each rule has only a finite number of instances with satisfiable bodies. As the dependency graph contains a finite number of nodes, the number of different non-empty strata is also finite.

7 Adding Interpreted Function Symbols

A practical ASP system has to have direct support for interpreted function symbols such as arithmetical operators. Otherwise, encoding problems involving arithmetics becomes cumbersome very quickly. In this work we take the approach that we add an interpretation function $\mathcal{I}$ that canonizes Herbrand terms. For example, with arithmetic addition we want to have $\mathcal{I}(+(5,2)) = 7$ where $+(5,2)$ is a Herbrand term formed from function symbols $+$, 5, and 2.

Definition 13. *Let P be a cardinality constraint program. Then, an* evaluator *is a function $\mathcal{I} : \mathbf{HU}(P) \to \mathbf{HU}(P)$.*

We also use $\mathcal{I}(F)$ to denote the formula that is obtained from F by evaluating all terms in it. For example, $\mathcal{I}(a(t_1, \ldots, t_n)) = a(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n))$. Next, we alter the definition of the instantiation of a rule so that all terms in it get evaluated.

Definition 14. *The* instantiation *of a rule $R = A \leftarrow C_1, \ldots, C_n$ is the set of rules:*

$$\mathbf{HI}(R, P) = \{\mathcal{I}(A\sigma) \leftarrow \mathcal{I}(I(C_1, P)\sigma), \ldots, \mathcal{I}(I(C_n, P)\sigma) \mid \\ \sigma \in \mathrm{Sub}(\mathrm{Var}(P), \mathbf{HU}(P))\} \tag{22}$$

In practice we can often save computational effort by evaluating a function as soon as we know its arguments.

We do not impose any requirements to the evaluation function E in this work. The most practical way to construct it is to have the logic programming tool to provide a number of built-in functions that implement the common arithmetic operations, and then suppose that any other function symbol has the Herbrand interpretation $E(x) = x$ for all $x \in \mathbf{HU}$. In the case that the user tries to use the built-ins in an undefined way, such as writing the term $2 + f(a)$, we have three possible ways to handle it: 1) we may revert back to the Herbrand interpretation; 2) add an explicit error term $\mathbf{e}$ and return that as an answer; or 3) add some high-level type checking into the tool and reject the whole program as erroneous whenever undefined operations occur.

8 Conclusions

We defined the class of cardinality constraint programs that allows the use of cardinality constraint and conditional literals. The language is a superset of normal logic programs. These constructs make it possible to express universal and existential quantifications in rule bodies. We also identified a decidable subset of it, ω-restricted programs, and showed how to formalize built-in functions.

A direct line of further research is to identify other types of aggregate literals that behave in a similar way to cardinality constraints, that is, aggregates that can be used as first-class literals in rules.

References

1. Christian Anger, Kathrin Konczak, and Thomas Linke. Nomore : A system for non-monotonic reasoning under answer set semantics. In *Proceedings of LPNMR'01*, pages 406–410, September 2001.
2. A. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 1:1–15, 1985.
3. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3), 374–425, 2001.
4. Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, and Gerald Pfeifer. System description: Dlv. In *Proceedings of LPNMR'01*, Vienna, Austria, September 2001. Springer-Verlag.
5. D. East and M Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of KI 2001*, pages 138–153, 2001.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP'88*, pages 1070–1080. The MIT Press, August 1988.
7. David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In *Proceedings of ILP'91*, pages 387–401. MIT Press, 1991.
8. Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.
9. Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
10. Tommi Syrjänen. Omega-restricted logic programs. In *Proceedings of LPNMR'01*, Vienna, Austria, September 2001. Springer-Verlag.
11. Tommi Syrjänen. Logic programming with cardinality constraints. Research Report A 86, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 2003.
12. Tommi Syrjänen and Ilkka Niemelä. The Smodels system. In *Proceedings of LP-NMR'01*, Vienna, Austria, September 2001. Springer-Verlag.