The Cardinality Operator: A new Logical Connective for Constraint Logic Programming

Pascal Van Hentenryck[1]
Yves Deville[2]

Technical Report No. CS-90-24
October 1990

[1]Computer Science Dept., Brown University, Providence, RI 02912
[2]University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium

The Cardinality Operator: A new Logical Connective for Constraint Logic Programming

Pascal Van Hentenryck
Brown University, Box 1910,
Providence, RI 02912
Email: pvh@cs.brown.edu

Yves Deville
University of Namur, 21 rue Grandgagnage
B-5000 Namur (Belgium)
Email: yde@info.fundp.ac.be

Abstract

This paper is an attempt to increase the (operational) expressiveness and efficiency of Constraint Logic Programming (CLP). We propose a programming abstraction, the cardinality operator, which provides CLP users with a new way of combining (primitive) constraints to build non-primitive ones. Declaratively, the cardinality operator is a new logical connective. Operationally, it implements a principle well-known in Artificial Intelligence and Operations Research, "*Infer simple constraints from difficult ones*", and is based on constraint-solving, constraint-entailment, and simple arithmetic reasoning. The cardinality operator is independent of the computation model, subsumes a number of control mechanisms and constraints that were sometimes introduced in a rather ad-hoc manner, and can be seen as a new relational combinator in the *ask & tell* framework. Programming examples and computation results are given, showing the versatility and efficiency of the operator.

1 Introduction

Constraint Logic Programming [18, 6] is the idea of replacing unification, the basic operation of Logic Programming (LP) language, by constraint-solving over a computation domain. Many CLP languages have been designed [6, 31, 11, 19] and implemented in recent years and computation domains include Linear Rational (resp. Real) Arithmetics [6, 19, 30], Boolean Algebra [6, 2], Presburger Arithmetics [31], and Finite Domains [28]. CLP languages have also been applied to many problems including combinatorial optimization (e.g. [27]), hardware design (e.g. [26]), and decision-support systems (e.g. [17, 1]).

The hypothesis underlying our research is the adequacy of CLP languages for the solving of combinatorial problems thanks to their unique combination of constraint-solving, nondeterminism, and relational form. In many cases, CLP languages enable to produce solutions to combinatorial problems, comparable in efficiency to procedural languages, yet requiring

a much shorter development time.

However, for other problems, existing CLP languages turn out to be not flexible or expressive enough (from an operational standpoint) to accommodate the pruning techniques that one would use in a procedural language. As a consequence, CLP languages might lose efficiency and turn out to be not practical for these problems.

The purpose of our research is thus the identification of suitable abstractions that enables, at the same time, a declarative statement of the problem and an operational behaviour matching the pruning techniques best appropriate for the problem at hand.

This paper describes a step towards that goal. We consider one of the main problems that a CLP programmer faces when constructing his program: the definition of non-primitive constraints, i.e. constraints that are not primitives of the language. In CLP, there are mainly two ways of combining primitive constraints to construct non-primitive constraints: using conjunctions of constraints (in a single clause possibly using recursion) or disjunctions of constraints (in separated clauses of a procedure possibly using recursion). Expressiveness is certainly not an issue but there is a significant difference between both alternatives from a computational standpoint. Indeed conjunctions of constraints are handled in a deterministic way, achieve pruning, and hence are perfectly appropriate. Disjunctions of constraints introduce nondeterminism and only prune the search space once a choice has been made. As a consequence, the behaviour obtained through disjunctions might be far away from the intended operational behaviour of the CLP programmer. As a matter of fact, in fields like Operations Research or Artificial Intelligence, constraints are rarely used to make choices, which are postponed until no information can possibly be deduced, but they are rather expected to reduce actively [13] the search space as soon as possible. Moreover a constraint, too difficult to be checked for consistency with other constraints, is handled locally and used to derive simpler constraints. It is only when all possible information have been deduced that a choice is taken and that choice can of course be guided by the constraints. This principle "*Infer simple constraints from difficult ones*" is actually the basic principle behind the design of CLP over Finite Domains, as clearly shown in [29], and would be appropriate to build non-primitive constraints as well.

Let us illustrate the point on a constraint appearing in a real problem [9]. Assume that the CLP programmer faces the constraint `atmost(Nb,[`X_1`,...,`X_n`],Val)` which holds if at most `Nb` elements X_i are equal to `Val`. A natural implementation of the constraint would be as follows.

```
atmost(Nb,[],Val).
atmost(Nb,[V|Vs],V) :-
     Nb > 0,
     Nb1 is Nb - 1,
     atmost(Nb1,Vs,V).
atmost(Nb,[V|Vs],Val) :-
     V ≠ Val,
     atmost(Nb,Vs,Val).
```

This implementation is however nondeterministic. A goal $\leftarrow$ `atmost(1,[X1,...,X3],4)` has three solutions, namely

$$X_1 = 4 \;\&\; X_2 \neq 4 \;\&\; X_3 \neq 4$$
$$X_1 \neq 4 \;\&\; X_2 = 4 \;\&\; X_3 \neq 4$$
$$X_1 \neq 4 \;\&\; X_2 \neq 4 \;\&\; X_3 = 4.$$

The expected operational behaviour of the constraint in the application (the one that the programmer would use in a procedural language) is in fact deterministic and can be described in the following way. The constraint has to check that, at any time of the computation, the accumulated constraints do not entail more than `Nb` elements X_i to be equal to `Val`. Moreover, if exactly `Nb` elements X_i are entailed to be equal to `Val`, then all other elements must be forced to be different from `Val`. Hence the constraint is expected to check (local) consistency with the accumulated constraints and to infer simpler constraints as soon as possible.

The above behaviour cannot be achieved through a conjunction of constraints of the form X_i = `Val` or $X_i \neq$ `Val`. The only way to achieve the intended behaviour would be to consider more basic variables (e.g. Boolean variables) and to express the constraints in terms of them. But this recasting of the problem substantially increases the number of variables, the number of constraints, and hence the search space to explore [12, 28].

The inability to obtain the desired operational behaviour for non-primitive constraints has led to a variety of control mechanisms such demons, forward and lookahead declarations, conditional propagation, and a number of primitive constraints (such as the `atmost` constraint) which avoid stating non-primitive constraints using disjunctions but might seem rather ad-hoc for someone external to the applications.

On the other hand, the *ask & tell* framework [22], stemming from research in concurrent logic programming [23], has generalized the CLP framework with the notion of constraint entailment [20, 11, 22]. The notion of constraint entailment seems fundamental in the building of non-primitive constraints but the generalized framework is still not sufficient in itself to subsume all other mechanisms and constraints mentioned above.

This paper introduces the cardinality operator to offer a new way to define non-primitive constraints. The cardinality operator is an abstraction allowing for a systematic construction of non-primitive constraints achieving the intended operational behaviour. It also encompasses the above control mechanisms and constraints. From a declarative standpoint, it is a new logical connective (Section 3). From an operational standpoint, it is based on constraint solving, constraint entailment, and arithmetic reasoning. It implements the principle "*Infer simple constraints from difficult ones*" and can also be viewed as a new combinator for the *ask & tell* framework (Section 4). It is independent of the computation domain and can be used for a variety of combinatorial problems (Section 5). Finally, it preserves the efficiency of the above mechanisms while allowing for order of magnitude speed-ups when they do not apply (Section 6).

2 Overview of Constraint Logic Programming

We start by giving an overview of Constraint Logic Programming following mainly [18]. This sets up the relevant background and fixes the notation used in the paper.

Constraint Logic Programming is a generalization of Logic Programming where unification, the basic operation of LP languages, is replaced by the more general concept of constraint-solving over a computation domain.

Syntactically, a CLP program can be seen as a finite set of clauses of the form

$$H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m$$

where $H, B_1, \ldots, B_m$ are atoms and $c_1, \ldots, c_n$ are constraints. A goal in a CLP language is simply a clause without head.

The declarative semantics of a CLP language can be defined either in terms of logical consequences or in an algebraic way. An answer to a CLP goal is no longer a substitution but rather a conjunction of constraints $c_1, \ldots, c_n$ such that

$P, \mathcal{T} \models (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G)$ (logical version)
$P \models_{\mathcal{S}} (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G)$ (algebraic version)

where P is a program, $\mathcal{S}$ is a structure, $\mathcal{T}$ is the theory axiomatizing $\mathcal{S}$, and $(\forall)(F)$ represents the universal closure of F.

The CLP theory [18] requires that the structure $\mathcal{S}$ be solution-compact and imposes conditions on the relationships on $\mathcal{T}$ and $\mathcal{S}$. The rest of the presentation can be read from a logic or algebraic point of view and we use the notation $\mathcal{D} \models$ to denote that fact.

The operational semantics of CLP amounts to replacing unification by constraint-solving. It might be defined by considering configurations of the form $\langle G, \sigma \rangle$ or σ where G is a conjunction of atoms and σ is a consistent conjunction of constraints. The transition rules between configurations can be given following an approach similar to [22]. In the following, we assume that P denotes a program, G a body and σ a conjunction of constraints. These are possibly subscripted.

True: The atom *true* simply leads to a terminal configuration.

$$\langle true\ ,\ \sigma \rangle \longmapsto \sigma$$

Goal Reduction: An atomic goal can be reduced to the body of a clause if the accumulated constraints are consistent with the new constraints.

$$\frac{H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m \in P \qquad \mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A)}{\langle A, \sigma \rangle \longmapsto \langle B_1 \wedge \ldots \wedge B_m\ ,\ \sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A \rangle}$$

In the above transition rule, $(\exists)(F)$ denotes the existential closure of F and we assume that the clause has been renamed with brand new variables.

Conjunction: If any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the accumulated constraints are updated accordingly.

$$\frac{\langle G_1 \, , \, \sigma \rangle \longmapsto \langle G_1' \, , \, \sigma' \rangle}{\begin{array}{l}\langle G_1 \,\&\, G_2 \, , \, \sigma \rangle \longmapsto \langle G_1' \,\&\, G_2 \, , \, \sigma' \rangle \\ \langle G_2 \,\&\, G_1 \, , \, \sigma \rangle \longmapsto \langle G_2 \,\&\, G_1' \, , \, \sigma' \rangle\end{array}}$$

$$\frac{\langle G_1 \, , \, \sigma \rangle \longmapsto \sigma'}{\begin{array}{l}\langle G_1 \,\&\, G_2 \, , \, \sigma \rangle \longmapsto \langle G_2 \, , \, \sigma' \rangle \\ \langle G_2 \,\&\, G_1 \, , \, \sigma \rangle \longmapsto \langle G_2 \, , \, \sigma' \rangle\end{array}}$$

The rules could be simplified at this point but the above presentation is better suited to understand the relationships between the cardinality operator and constraint-solving.

The actual operational semantics of the language can be defined in terms of its success, divergence, and failure sets. Let $\stackrel{*}{\longmapsto}$ denote the reflexive and transitive closure of $\longmapsto$. Also we say that a configuration γ diverges in program P if there exists an infinite sequence of transitions $P \vdash \gamma \longmapsto \gamma_1 \longmapsto \ldots \longmapsto \gamma_i \longmapsto \ldots$
The success, floundering, and divergence sets are defined in the following way.

$$\begin{array}{lcl} SS[P] & = & \{\langle G, \sigma \rangle \mid P \vdash \langle G, \sigma \rangle \stackrel{*}{\longmapsto} \sigma'\} \\ DS[P] & = & \{\langle G, \sigma \rangle \mid \langle G, \sigma \rangle \text{ diverges in P}\} \end{array}$$

The failure set can now be defined in terms of the above two sets.

$$FS[P] \quad = \quad \{\langle G, \sigma \rangle \mid \langle G, \sigma \rangle \notin SS[P] \cup DS[P]\}$$

Another semantic definition can be given to capture the results of the computation.

$$RES[P]\langle G, \sigma \rangle \quad = \quad \{\sigma' \mid P \vdash \langle G, \sigma \rangle \stackrel{*}{\longmapsto} \sigma'\}$$

In order to achieve the above semantics, the CLP language should be embedded with a constraint-solver that is complete which means that, given a conjunction of constraints σ, the constraint-solver should return *true* if $\mathcal{D} \models (\exists)(\sigma)$ and $false$ otherwise.

3 Syntax and Declarative Semantics

This section describes the syntax and declarative semantics of the cardinality operator. The cardinality operator allows for a new kind of formulas, cardinality formulas, with the following formation rule.

Definition 1 [Extension of the formation rules] If $\psi_1, \ldots, \psi_n$ are formulas and l, u are terms, then $\#(l, u, [\psi_1, \ldots, \psi_n])$ is a formula.

In cardinality formulas, l, u are called the bounds. We extend the syntax of CLP programs by allowing $B_1, \ldots, B_n$ in the body of clauses to be either atoms or cardinality formulas.

The declarative semantics is given by adding an interpretation rule in the definition of satisfiability.

Definition 2 [Extension of Satisfiability] If I is an interpretation and θ is a variable assignment, the truth value of a cardinality formula $\#(l, u, [\psi_1, \ldots, \psi_n])$ wrt I and θ is *true* if $\theta(l)$ and $\theta(u)$ are integers and the number of formulas ψ_i assigned to *true* wrt I and θ is not less than $\theta(l)$ and not more than $\theta(u)$.

It is interesting to note at this point that cardinality formulas are quite expressive. A conjunction $\psi_1 \wedge \ldots \wedge \psi_n$ can be expressed by $\#(n, n, [\psi_1, \ldots, \psi_n])$, a disjunction $\psi_1 \vee \ldots \vee \psi_n$ by $\#(1, n, [\psi_1, \ldots, \psi_n])$, and a negation $\neg\psi$ by by $\#(0, 0, [\psi])$. Implication can be expressed in terms of disjunction and negation and equivalence in terms of two implications.

However the reader should keep in mind that the cardinality operator does not aim at solving the general problem of negation and disjunction in logic programming. It is intended as an abstraction to build, in a systematic way, non-primitive constraints achieving an operational behaviour that is appropriate in practice. The fact that the abstraction can be given a natural declarative semantics is a strength of the extension and fits well in the spirit of logic programming.

4 Operational Semantics

This section describes the operational semantics of the cardinality operator. As mentioned, the purpose of the cardinality operator is the expression of non-primitive constraints implementing the principle "*Infer simple constraints from difficult ones*". Hence we impose syntactical restrictions on the formulas and terms appearing in the cardinality formulas. In the first subsection, we give the basic semantics which enforces the most severe restrictions. The next sections relax most of the restrictions. We conclude the section by discuting a meta-level version of the operator and some syntactic abbreviations.

4.1 The Basic Operational Semantics

In the basic semantics, the formulas $\psi_1, \ldots, \psi_n$ and the terms l, u in $\#(l, u, [\psi_1, \ldots, \psi_n])$ are required to be respectively (primitive) constraints and integers. The operational semantics of the cardinality operator is defined in terms of transition rules.

Trivial Satisfaction: If $l \leq 0$ and u is greater or equal to the number of constraints $c_1, \ldots, c_n$, then $\#(l, u, [c_1, \ldots, c_n])$ is trivially satisfied.

$$\frac{l \leq 0 \ \wedge \ n \leq u}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma}$$

Positive Satisfaction: A formula $\#(n, u, [c_1, \ldots, c_n])$ with $n \leq u$ can only be satisfied if the conjunction $c_1 \wedge \ldots \wedge c_n$ is consistent with the accumulated constraints.

$$\frac{l \leq u \ \wedge \ l = n \qquad \mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n)}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge c_1 \wedge \ldots \wedge c_n}$$

Negative Satisfaction: A formula $\#(l, 0, [c_1, \ldots, c_n])$ with $l \leq 0$ can only be satisfied if the conjunction $\neg c_1 \wedge \ldots \wedge \neg c_n$ is consistent with the accumulated constraints.

$$\frac{l \leq u \ \wedge \ u = 0 \qquad \mathcal{D} \models (\exists)(\sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n)}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n}$$

The above three rules make up the basic cases for the cardinality operator. Two of them allow for the inference of primitive constraints, and hence prune the search space with the help of the transition rule for conjunction. Note that the third rule requires that the negation of a constraint be itself a constraint.

Positive Reduction: When a constraint c_i is entailed by the accumulated constraints, the cardinality formula can be simplified by dropping the constraint and decrementing the bounds.

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow c_i) \qquad 0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \langle \#(l-1, u-1, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]) \, , \, \sigma \rangle}$$

The condition on l and u forces the rule to be mutually exclusive with the three satisfaction rules.

Negative Reduction: When the negation of a constraint c_i is entailed by the accumulated constraints (i.e. c_i inconsistent with σ), the cardinality formula can be simplified by dropping the constraint.

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow \neg c_i) \qquad 0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \langle \#(l, u, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]) \, , \, \sigma \rangle}$$

The above two rules achieve progress towards the satisfaction rules by reducing the number of constraints and (possibly) the bounds.

The computation with the cardinality operator may now flounder as none of the constraints can be decided upon (for entailment) wrt the accumulated constraints. The configurations have to be generalized to include *flounder* as a new terminal and one more transition rule has to be defined.

Floundering: The cardinality operator flounders if there is no constraint c_i such either c_i or its negation $\neg c_i$ is entailed by the accumulated constraints and none of the satisfaction rules apply.

$$\frac{\begin{array}{l}\mathcal{D} \not\models (\forall)(\sigma \Rightarrow c_j) \quad \text{for all } 1 \leq j \leq n \\ \mathcal{D} \not\models (\forall)(\sigma \Rightarrow \neg c_j) \quad \text{for all } 1 \leq j \leq n \\ 0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0\end{array}}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \mathit{flounder}}$$

The operational semantics of the CLP language now needs one more set, the floundering set, defined as follows.

$$FLS[P] \quad = \quad \{\langle G, \sigma \rangle \mid \langle G, \sigma \rangle \stackrel{*}{\longmapsto} \mathit{flounder}\}$$

The failure set has to be redefined in terms of the above three sets.

$$FS[P] \quad = \quad \{\langle G, \sigma \rangle \mid \langle G, \sigma \rangle \notin SS[P] \cup FLS[P] \cup DS[P]\}$$

It should be mentioned that the cardinality operator introduces a new kind of incompleteness in the language. This is intentional as explained in the motivation and perfectly acceptable as it enables to achieve the expected pruning.

4.2 Expressions in Bounds

If the CLP language is endowed with a constraint-solver over integers (or finite sets of integers), then it is possible to relax the requirement that l and u be integers and to allow instead for any expression over integers. The gain here is the ability to reason about the number of true constraints in a set (or multiset) using cardinality formulas of the form $\#(V, V, [c_1, \ldots, c_n])$ where V is a variable over integers. The above cardinality formula actually constrains V to be the number of c_i which are true and this variable can be used in other (primitive or non-primitive) constraints.

Expressions in Bounds require a slight adaptation of existing rules and the addition of a new one, giving the following semantics.

Trivial Satisfaction:

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow (l \leq 0 \ \wedge \ n \leq u))}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma}$$

Positive Satisfaction:

$$\frac{\begin{array}{l}\mathcal{D} \models (\forall)(\sigma \Rightarrow (l \leq u \ \wedge \ l = n)) \\ \mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n)\end{array}}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge c_1 \wedge \ldots \wedge c_n}$$

Negative Satisfaction:

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow (l \leq u \ \wedge \ u = 0)) \quad \mathcal{D} \models (\exists)(\sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n)}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \sigma \wedge \neg c_1 \wedge \ldots \wedge \neg c_n}$$

For the application of the reduction and floundering rules, the condition on l and u has only to be consistent with the accumulated constraints. Note that such a condition ensures these rules to be mutually exclusive with the satisfaction rules.

Positive Reduction:

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow c_i) \quad \mathcal{D} \models (\exists)(\sigma \ \wedge \ (0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0))}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \langle \#(l-1, u-1, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]) \, , \, \sigma \rangle}$$

Negative Reduction:

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow \neg c_i) \quad \mathcal{D} \models (\exists)(\sigma \ \wedge \ (0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0))}{\langle \#(l, u, [c_1, \ldots, c_i, \ldots, c_n]) \, , \, \sigma \rangle \longmapsto \langle \#(l, u, [c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n]) \, , \, \sigma \rangle}$$

These transition rules does not constrain the bound expressions l and u to any value. The following transition rule adds bound constraints in the accumulated constraints.

Bound Constraints:

$$\frac{\mathcal{D} \not\models (\forall)(\sigma \Rightarrow l \leq u \ \wedge \ l \leq n \ \wedge \ 0 \leq u) \quad \mathcal{D} \models (\exists)(\sigma \ \wedge \ l \leq u \ \wedge \ l \leq n \ \wedge \ 0 \leq u)}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \ \longmapsto \ \langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \wedge l \leq u \wedge l \leq n \wedge 0 \leq u \rangle}$$

The first condition ensures that the added constraints are not redundant (hence avoiding an infinite application of the rule). The second condition ensures the consistency of the accumulated constraints. The bound constraints together with the reduction rules enforce maximal constraints on the bound expressions.

Floundering:

$$\frac{\mathcal{D} \not\models (\forall)(\sigma \Rightarrow c_i) \quad \mathcal{D} \not\models (\forall)(\sigma \Rightarrow \neg c_i) \quad \mathcal{D} \models (\exists)(\sigma \ \wedge \ (0 < l < n \ \wedge \ l \leq u \ \ \vee \ \ 0 < u < n \ \wedge \ l \leq 0))}{\langle \#(l, u, [c_1, \ldots, c_n]) \, , \, \sigma \rangle \ \longmapsto \ \mathit{flounder}}$$

4.3 Imbricated Cardinality

A further relaxation of the conditions is to allow cardinality formulas inside cardinality formulas. Such a relaxation is a straightforward extension of the above operational semantics with variables in bounds. The basic idea is to rewrite the formula

$$\#(l, u, [\psi_1, \ldots, \psi_k, \ldots, \psi_n])$$

where ψ_k is a cardinality formula $\#(l', u', [\phi_1, \ldots, \phi_m])$, as the conjunction

$$\#(V, V, [\phi_1, \ldots, \phi_m]) \ \& \ \#(l, u, [\psi_1, \ldots, l' \leq V \leq u', \ldots, \psi_n])$$

where V is a brand new variable.

The two formulas are equivalent and achieve the same pruning. The overhead (i.e introduction of new variables) is linear with the number of imbricated cardinality constraints.

4.4 Meta-level Extension

So far, the cardinality operator requires to know the constraints $c_1, \ldots, c_n$ at compile time. Programming with the cardinality operator can be simplified in many cases by allowing a more dynamic version of the construct. Essentially, it would mean that the programmer be able to write generic non-primitive constraints that do not require the knowledge of the constraints at compile time. Although such an extension does not add any expressive power in theory, it does simplify the programming task.

The declarative semantics of the language can then be specified using an approach like Hilog [3]. Also, the operational semantics imposes implicit (ask) constraints on the third argument of the operator, namely that the argument should be a list of constraints, or in symbols

$$\mathcal{D} \models (\forall)(\sigma \Rightarrow L = [c_1, \ldots, c_n])$$

where σ are the accumulated constraints and L is the third argument of the cardinality operator. Note that the cardinality operator (or more precisely its meta-level extension) can flounder if the above condition is not met.

4.5 Syntactical Abbreviations

We have seen previously that the cardinality operator subsumes (from a declarative standpoint) the other logical connectives. In the following, we feel free to use the logical connective version instead of the cardinality operator keeping in mind that the intended operational behaviour is given by the above transition rules. We also use $*$ instead of a bound when it is not relevant to the (non-primitive) constraint at hand. It can be seen as a 0 in a lower bound, and as n in an upper bound.

5 Program Examples

In this section, we consider various simple constraints that can be stated in terms of the cardinality operator. Examples are drawn from different computation domains to illustrate the generality of the extension.

5.1 Finite Domains

Finite Domains is a computation domain where there is a large potential for the cardinality operator. This comes from the large variety of problems that can be solved in that framework which require various kinds of non-primitive constraints. In the following, we give several examples of constraints on Finite domains and suggest several other uses.

Disjunctive Constraints

In scheduling, tasks using the same resource cannot be scheduled at the same time. If S_1, S_2 represent starting dates of two tasks and D_1, D_2 represent their durations, the disjunctive constraint can be stated as follows [10, 28].

```
disjunction(S1,D1,S2,D2) :-
     S1 + D1 ≤ S2.
disjunction(S1,D1,S2,D2) :-
     S2 + D2 ≤ S1.
```

In the program, S_1, S_2, and possibly D_1, D_2, are expected to be Finite Domains variables or natural numbers. The constraint is implemented as a disjunction of constraints and hence is nondeterministic. A more suitable approach would be to deduce information from the constraint and to postpone the choices until no information is deducible from the constraints. The above constraint can be used in two (symmetric) ways to achieve this pruning:

- if the maximal start date of S_2 is smaller than the minimal start date of S_1 added to D_1, then task 2 cannot be scheduled after task 1;
- if the maximal start date of S_1 is smaller than the minimal start date of S_2 added to D_2, then task 1 cannot be scheduled after task 2;

Using the cardinality operator, the above handling is achieved by the following program.

```
disjunction(S1,D1,S2,D2) :-
     #(1,*,[S1 + D1 ≤ S2, S2 + D2 ≤ S1]).
```

The operational behaviour of the above constraint makes sure that at least one of the two possibilities is consistent with the accumulated constraints. Moreover if only one is consistent, the appropriated constraint is added to the accumulated constraints.

The above handling, and more sophisticated ones based on the same idea, might dramatically improve the efficiency of scheduling algorithms.

The atmost Constraint

The atmost constraint used in the introduction can be implemented as follows using the meta-level facility to build constraints dynamically.

```
atmost(Nb,L,Val) :-
     collect(L,Val,Lcstrs),
     #(*,Nb,Lcstrs).

collect([],Val,[]).
collect([X|Xs],Val,[X = Val|Lcstrs]) :-
     collect(Xs,Val,Lcstrs).
```

Intuitively, the collect predicate constructs the list of constraints and the cardinality operator uses the list to make sure that at most Nb of them are true. Given a goal "← atmost(1,[X_1,...,X_3],4)", the above program generates the cardinality formula

#(*,1,[X_1=4,X_2=4,X_3=4]).

Hence, as soon as one of the variables is fixed to be equal to 4 by the accumulated constraints, the two other variables are constrained to be different from 4.

The element Constraint

The element constraint has turned out to be instrumental in solving many combinatorial optimization problems (e.g. [8]). Declaratively, element(X,[v_1,...,v_n],E) holds if E is equal to v_X ($1 \leq X \leq n$). Operationally, $v_1,\ldots,v_n$ are assumed to be integers (non necessarily distinct), X and E are assumed to be domain variables. The predicate enforces a dependency between $X \in \{1,\ldots,n\}$ and $E \in \{v_1,\ldots,v_n\}$ so that each time the domain of X is modified, the domain of E is updated accordingly and vice versa. The operational semantics can be characterized by a conjunction of constraints, with one constraint of the following form for each different value $v \in \{v_1,\ldots,v_n\}$

$$E = v \Leftrightarrow X \in \{i_1,\ldots,i_m\}$$

where $v_{i_1},\ldots,v_{i_m}$ are all the elements equal to v. These constraints can be generated using the collect and state approach used for the atmost constraint.

Miscellaneous

As shown above, symbolic constraints that were previously considered primitives of the language can be defined declaratively in terms of the cardinality operator, achieving a similar operational pruning. The cardinality operator can be used as well to implement lookahead declarations [28]. These do not need to be seen as primitive control mechanisms but can be provided as syntactic sugar to reduce the programming effort. The above example also demonstrates that constraint solving and constraint entailment on a small set of constraints in conjunction with the cardinality operator are sufficient provide a rich language.

5.2 Linear Rational (Real) Arithmetics

Linear Rational (Real) Arithmetics is another computation domain that can possibly benefit from the cardinality operator. In many applications such as circuit design [16, 14] or decision-support systems [17, 1], users encounter non-linear constraints and approach them using piecewise linear approximations. This often results in definitions using disjunctions and the cardinality operator offers a more active way of handling them. As a simple example, consider a piecewise linear approximation of a function with two variables. It is quite common to find code of the following form.

```
p_linear(X,Y) :-
     Bx1 ≤ X < Bx2,
     By1 ≤ Y < By2.
p_linear(X,Y) :-
     Bx2 ≤ X < Bx3,
     By2 ≤ Y < By3.
...
p_linear(X,Y) :-
     Bxn-1 ≤ X < Bxn,
     Byn-1 ≤ Y < Byn.
```

The search space could be reduced by devising a constraint that makes sure that

- at least one of the cases still applies wrt the accumulated constraints;
- as soon as X (resp. Y) is entailed to be within an interval, Y (resp. X) is constrained to be in the corresponding interval.

This constraint can be implemented using the following program.

```
piecewise_linear(X,Y,[_,_]).
piecewise_linear(X,Y,[Bx1,By1,Bx2,By2|Bs]) :-
     Bx1 ≤ X < Bx2 ⇔ By1 ≤ Y < By2,
     piecewise_linear(X,Y,[Bx2,By2|Bs]).
```

The goal "← `piece_linear(X,Y,[Bx1,By1,...,Bxn,Byn])`" then achieves the intended behaviour.

As might be seen, the cardinality operator subsumes conditional propagation (which is similar to ask constraints [22]) used to achieve the intended behaviour in [14]. The cardinality operator, because of its relational form, gives a more compact definition (e.g. the size programs using conditional propagation can quickly become explosive) and a more efficient implementation (e.g. constraints are shared).

5.3 Herbrand

Constraint-solving over Herbrand has turned out to be instrumental for applications in circuit design as examplified by [25, 24]. The techniques used here are basically demon-driven computation that could be obtained by concurrent logic programming [23] and demon declarations [11]. The cardinality operator can be used to achieve the same behaviour, once again in a more general setting. We illustrate the point by showing the implementation of an and-gate with four signals [24].

```
and(X,Y,Z) :-
     X = 0 ∨ Y = 0 ⇒ Z = 0,
     Z = 1 ⇒ X = 1 ∧ Y = 1,
     X = 1 ⇒ Y = Z,
     Y = 1 ⇒ X = Z,
     X = d ∨ X = dnot ⇒ Z = X ∧ Y = 1,
     Y = d ∨ Y = dnot ⇒ Z = Y ∧ X = 1.
```

The cardinality operator allows for a more succinct representation (e.g. cases can be merged), and possibly for a more efficient implementation (e.g. if constraint-solving over disequations is used).

6 Computation Results

This section is intended to show the efficiency of the cardinality operator. This is done in two steps. In the first step, we show that for some problems the cardinality operator can produce arbitrarily large speed-ups. In a second step, we show that non-primitive constraints built with the cardinality operator are comparable to the efficiency one would obtain by implementing the constraint as a primitive of the language.

6.1 Speedups

We present a simple example, the magic series problem (see e.g., [28]), to give the reader an idea of the kind of speedups that can be produced by the cardinality operator. Industrial applications including scheduling and planning problems have shown similar behaviours but are not in the scope of this paper.

Let S = (s_0, s_1, ..., s_n) be a non-empty finite series of non-negative integers. We take the convention of numbering its elements from zero. The series S is said *magic* if and only if there are s_i occurrences of i in S, for each integer i ranging from 0 to n. In an equivalent way

$$\forall\, 0 \leq i \leq n : s_i = card\{j : 0 \leq j \leq n \text{ and } s_j = i\}$$

where $card(S)$ denotes the cardinality of set S. For instance, for $n = 3$, the series $(1, 2, 1, 0)$ is magic.

```
magic(N,L) :-
  N1 is N + 1,
  create_vars(N1,L,0,N),
  occurrences(L,0,L),
  labeling(L).

occurrences([],Val,L).
occurrences([F|T],Val,L) :-
  occur(F,L,Val),
  V1 is Val + 1,
  occurrences(T,V1,L).
```

Figure 1: A Magic Series Program

```
occur(Nb,L,Val) :-
  collect(L,Val,Eqs),
  #(Nb,Nb,Eqs).

collect([],Val,[]).
collect([V|Vs],Val,[V = Val|Eqs]) :-
  collect(Vs,Val,Eqs).
```

Figure 2: The occur Predicate using the Cardinality Operator

A solution to this problem is presented in Figure 1. Procedure `create_var(N,L,Low,Up)` holds if `L` is a list of `N` domain-variables ranging from `Low` to `Up`. Procedure `labeling(L)` assigns a value to the elements of `L`. These straightforward procedures are not shown here.

Figure 2 depicts the implementation of the occur predicate using the cardinality operator.

Figure 3 shows the standard implementation of the occur predicate with a CLP language over Finite Domains (e.g., the CHIP language). It is taken from [28]. Note that, within this approach, the occurrences predicates generate occur predicates which are delayed until their first argument is ground. Once an occur predicate is selected, it makes some choices on the problem variables. This formulation, although dramatically better than a standard backtracking approach, suffers from the drawbacks mentioned in the introduction. The occur constraints are used for making choices and not for pruning the search space actively.

Table 1 presents the computation times of both programs to compute all solutions for various values of N[1]. The standard CLP program (e.g., the CHIP program) is denoted CLP while the program with the cardinality operator is denoted by CCLP. Both programs have

[1]The results for finding one solution are fully consistent with the ones presented.

```
delay occur(ground,any,any).
occur(0,Val,L) :-
   outof(Val,L).
occur(Nb,Val,[Val|L]) :-
   Nb > 0,
   Nb1 is Nb - 1,
   occur(Nb1,Val,L).
occur(Nb,Val,[V|L]) :-
   Nb > 0,
   Val ≠ V,
   occur(Nb,Val,L).
```

Figure 3: The occur Predicate in a Standard CLP Language

been executed on a SUN 3/60 with our compiler. The times are given in seconds. As can be seen for the table, the speedup produced by the cardinality operator increases exponentially with N. For N = 8, we already have a speedup of 140 !

The above programs can be improved by adding a redundant constraint stating that the summation of the elements of L must be equal to N+1. Let us denote CLP_A and $CCLP_A$ the above programs with that additional constraint. Table 1 also gives the results for these improved programs. Interestingly enough, $CCLP_A$ still produces a speedup over CLP_A increasing exponentially with N. Moreover CCLP is more efficient than CLP_A as soon as $N \geq 9$. Of course $CCLP_A$ is the most efficient program. Note however that the gain of $CCLP_A$ over CCLP is small compared to gain of CLP_A over CLP. This clearly demonstrates the pruning capability and efficiency of the cardinality operator. It also demonstrates that with CCLP the need for additional and redundant constraints is not as crucial as with CLP.

6.2 Comparison with Primitive Constraints

Implementations of non primitive constraints through the cardinality operator are comparable in efficiency to builtin implementations of these constraints. In order to demonstrate that claim, we compare two algorithms solving car-sequencing problems [9] with our compiler: the first one with the atmost constraint implemented as an additional primitive of the language and the second one with the atmost constraint implemented through the cardinality operator. The overhead produced by the cardinality operator was around 30 % for problems involving hundreds of cars. This clearly demonstrates the efficiency of the operator. Moreover, the introduction of the cardinality operator induces a very small overhead when added to a CLP language and not used in the solving of a problem. The additional cost comes from the need to check if some constraints have to be considered for entailment.

Nb	CLP	CCLP	CLP/CCLP	CLP_A	$CCLP_A$
4	0.38	0.32	1.18	0.20	0.28
5	1.82	0.76	2.39	0.46	0.50
6	11.56	1.64	7.04	1.06	0.88
7	88.38	3.18	27.79	2.30	1.48
8	800.78	5.70	140.48	4.90	2.36
9	??	9.62	??	10.40	3.62
10	??	15.36	??	21.92	5.46
11	??	23.56	??	46.16	7.94
12	??	34.82	??	101.84	11.30
13	??	49.98	??	202.84	15.78
14	??	70.32	??	427.72	21.54

Table 1: Results for the Magic Series Problem

7 Related Work

As can be seen from the text, this work builds on top, or relates to, many other work in Logic Programming. For instance, constraint entailment is borrowed from [20, 22, 11]. The underlying motivation behind the cardinality operator comes from demon-propagation, conditional propagation, forward & lookhead declarations and the primitive constraints of [11], as well as from real applications. The form of the operational semantics follows [22]. Related work include concurrent logic programming [23], the andorra-model [15], and meta-programming [3]. Finally, coroutining [5, 4, 7, 21] is, as always, the basic block for implementation issues.

An appropriate comparison would deserve more than a simple section and is beyond the scope of the paper. Moreover the relevance of the cardinality operator to concurrent (constraint) logic programming remains to be investigated.

8 Conclusions

In this paper, we have introduced the cardinality operator to improve the (operational) expressiveness and efficiency of CLP languages. The cardinality operator offers a new way of building non-primitive constraints from primitives of the language and subsumes many existing control mechanisms and constraints. The operator has a simple declarative semantics and implements, at the operational level, the well-known principle "*Infer simple constraints from difficult ones*". Programming examples and computational results have shown the versatility of the abstraction.

Future work will include the application of operator in other domains such as concurrent programming and Boolean Algebra as well as improvements of the implementation to remove part of the remaining overhead. The search for other operators and abstraction will of course remain an important topic.

Acknowledgments

We are grateful to Vijay Saraswat for showing us how to make it simple, to Thierry Le Provot for motivating the research, and the members of the CHIP team for discussions on this topic. The second author is supported by the Belgian National Fund for Scientific Research as a Research Associate.

References

[1] F. Berthier. Using CHIP to Support Decision Making. In *Actes du Seminaire 1988 - Programmation en Logique*, Tregastel, France, May 1988. CNET.

[2] W. Buttner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[3] W. Chen, M Kifer, and D.S Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. In *Proceedings of the North-American Conference on Logic Programming (NACLP-89)*, Cleveland, Ohio, October 1989.

[4] K.L. Clark and F. Mc Cabe. The Control Facilities of IC-PROLOG. In D. Mitchie, editor, *Expert systems in the micro electronic Age*, pages 122–149. Edinburgh University Press, 1979.

[5] A. Colmerauer. PROLOG II: Manuel de Reference et Modele Theorique. Technical report, GIA - Faculte de Sciences de Luminy, March 1982.

[6] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[7] M. Dincbas. The Metalog Problem-solving System, an Informal Presentation. In *Proceedings of Logic Programming Workshop*, pages 80–91, Debrecen, Hungary, July 1980.

[8] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a Cutting-Stock Problem in Constraint Logic Programming. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[9] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[10] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[11] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japon, December 1988.

[12] S. French. *Sequencing and Scheduling.* Mathematics and its Applications. Ellis Horwood Limited, Chichester, England, 1982.

[13] H. Gallaire. Logic Programming: Further Developments. In *IEEE Symposium on Logic Programming*, pages 88–99, Boston, July 1985. Invited paper.

[14] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. *Computers and Mathematics with Applications*, 20(9/10):45–56, 1990.

[15] S. Haridi. A Logic Programming Language based on the Andorra Model. *New Generation Computation*, 1990. To Appear.

[16] N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and Some Electrical Engineering Problems. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[17] T. Huynh and C. Lassez. A CLP($\Re$) Option Trading Analysis System. In *Fifth International Conference on Logic Programming*, Seattle, WA, August 1988.

[18] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[19] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[20] M.J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[21] L. Naish. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, Australia, 1985.

[22] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[23] E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 1990. To Appear.

[24] H. Simonis. Test Generation using the Constraint Logic Programming Language CHIP. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[25] H. Simonis and M. Dincbas. Using Logic Programming for Fault Diagnosis in Digital Circuits. Technical Report TR-LP-18, ECRC, December 1986.

[26] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, Munich, RFA, Octobre 1987.

[27] P. Van Hentenryck. A Logic Language for Combinatorial Optimization. *Annals of Operations Research*, 21:247–274, 1989.

[28] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[29] P. Van Hentenryck and Y. Deville. Operational Semantics of Constraint Logic Programming over Finite Domains. Technical report, CS Department, Brown University, 1990. (Forthcoming).

[30] P. Van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. In *International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 1990. Also ECRC internal Report IR-LP-2217.

[31] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.