Extending Constraint Logic Programming with Open Functions

Nikolay Pelov
Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
pelov@cs.kuleuven.ac.be

Maurice Bruynooghe
Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
maurice@cs.kuleuven.ac.be

ABSTRACT
The natural representation of solutions of finite constraint satisfaction problems is as a (set of) function(s) or relation(s). In (constraint) logic programming, answers are in the form of substitutions to the variables in the query. This results in a not very declarative programming style where a table has to be presented as a complex term. Recently, stable logic programming, also called answer set programming and abductive logic programming have been proposed as approaches supporting a more declarative style for solving such problems.

The approach developed in this paper is to extend the constraint domain of a constraint logic programming language with open functions, functions for which the interpretation is not fixed in advance. Their interpretation contains the solution of the problem. This enrichment of the constraint domain yields a language which is almost as expressive as abductive logic programming and is very well suited for expressing finite domain constraint satisfaction problems. Implementation requires only to extend the constraint solver of the underlying CLP language.

1. INTRODUCTION
For many problems, the most natural representation of their solution is as a (set of) table(s) or relation(s) (e.g. [23]). Consider for example the n-queens problem. A natural representation of its solution is as a table of facts $position(i, j)$ where a pair (i, j) defines the coordinates of a queen or as a function $position(i)$ where the function value for i defines the column position of the queen on row i. In the framework of (constraint) logic programming, solutions are substitutions for the variables in some goal. Hence a programmer using this paradigm is forced to choose a term representation for the solution table. This leads to a procedural programming style of generating a list of domain variables holding the positions of the different queens, of setting up the constraints between the different queens and finally of searching for a solution satisfying the constraints. The level of indirection in the representation results in a much less *declarative* program (e.g. [8]). It is harder to write, to understand and to maintain than in the case where a solution can be represented as a relation or a function. A typical CLP-program for solving the n-queens problem (note the double recursion) is as follows:

```
queens(Q,N) :- generate(Q,N,N), safe(Q), instantiate(Q).

generate([],0,_).
generate([X|T],M,N) :-
    M > 0, X in 1..N, M1 is M-1, generate(T,M1,N).

safe([]).
safe([X|T]) :- noAttack(X,1,T), safe(T).

noAttack(_,_,[]).
noAttack(X,D,[Y|Z]) :-
   X \= Y, abs(X-Y) \= D,
   D1 is D1 + 1, noAttack(X,D1,Z).

instantiate([]).
instantiate([X|T]) :- enum(X), instantiate(T).
```

A number of recent papers argue in favor of a new logic programming paradigm based on stable model semantics [14]. In [27], Marek and Truszczyński introduce Stable Logic Programming as a novel programming paradigm. The language is basically DATALOG extended with negation. Stable models of programs form a finite family of finite sets, hence the solutions to search problems can be represented as stable models of Stable Logic Programs. The programming style is to introduce constraints which restrict the stable models to the solutions. Lifschitz [25, 26] introduces the closely related notion of Answer Set Programming and explores its use in satisfiability planning [24]. Also Niemelä [28] makes a similar point, proposing stable models as a paradigm for constraint programming. The **smodels** system [29] is one of the most advanced implementations supporting this programming paradigm.

A more smooth extension of the basic logic programming paradigm is abductive logic programming [19]. Besides the program P, it distinguishes a set of abducibles A (names of predicates not defined in the program) and a set of integrity constraints IC. An abductive solver then searches for a set of atoms Δ for the abductive predicates such that all integrity constraints are satisfied. However abductive logic programming systems have complex inference rules (e.g.

the inference rules in [9, 13, 21, 22]). As a consequence, it is rather hard to understand and control their procedural behavior. Moreover a direct implementation as an extension of a logic programming system is far from obvious. Current implementations are meta-interpreters on top of a (constraint) logic programming system [9, 21, 22]. The meta-interpreter overhead is detrimental to their performance and their memory consumption.

This paper develops another approach. Integrity constraints are retained –their evaluation can be reduced to query evaluation by SLDNF– but abducibles are dropped. Instead, we introduce functions which are allowed to have an arbitrary interpretation, called *open functions*[1]. This retains most of the expressiveness of abductive logic programming while it results in a substantially simpler inference system, close to SLDNF. Indeed, the required modification is to enrich the constraint domain of a CLP language with open functions and to extend the constraint solver with inference rules to cope with the open functions. Modifying the constraint solver of an existing CLP language is perhaps easier than building an abductive system from scratch. It is part of our future research to find out.

The paper starts with recalling some of the preliminaries on which constraint logic programming is based. Section 3 describes how to extend constraint domains with open functions; it introduces an open function solved form and describes which rewrite rules to add to an existing solver to derive the open function solved form. Section 4 introduces the programming paradigm. How to adapt a CLP semantics to the extended language is also described in this section. Section 5 presents a proof procedure which is based on the SLDNF proof procedure of Apt and Doets and also shows an extension incorporating tabulation. Section 6 shows advanced examples illustrating the need for tabulation and negation. The paper ends with a discussion of related work and a conclusion.

2. PRELIMINARIES

A sequence of terms $s_1, \ldots, s_n$ is denoted as $\tilde{s}$ and we use $\tilde{s} = \tilde{t}$ as a shorthand for $s_1 = t_1, \ldots, s_n = t_n$.

To facilitate the introduction of open functions, we recall some basic notions about many-sorted signatures and structures (e.g. [35]). A *signature* Σ is a pair $\langle \mathcal{S}, \mathcal{F} \rangle$ where $\mathcal{S}$ is a set of *sorts* (*types*), $\mathcal{F}$ is a set of function and predicate symbols together with an arity function τ which maps function symbols to $\mathcal{S}^+$ and predicate symbols to $\mathcal{S}^*$. We write $f : \tilde{w} \to s$ when $\tau(f) = \tilde{w}, s$. A *constant* has a type $\to s$. A signature $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$ is *embedded* in $\Sigma' = \langle \mathcal{S}', \mathcal{F}' \rangle$ if $\mathcal{S} \subseteq \mathcal{S}'$, $\mathcal{F} \subseteq \mathcal{F}'$ and $\tau = \tau'|_{\mathcal{F}}$. Later on we use embeddings where $\mathcal{S} = \mathcal{S}'$.

A Σ-structure $\mathcal{D}$ defines an interpretation for a signature Σ. For each sort s it defines a domain D_s (the *universe* or *carrier* of sort s), for each function symbol $f : s_1, \ldots, s_n \to s$ a function $f^D : D_{s_1}, \ldots, D_{s_n} \to D_s$ and for each predicate symbol $p : s_1, \ldots, s_n$ a relation $p^{\mathcal{D}} \subseteq D_{s_1} \times \ldots \times D_{s_n}$. A structure $\mathcal{D}$ is a Σ-*reduct* of a structure $\mathcal{D}'$, denoted with $\mathcal{D} = \mathcal{D}'|_\Sigma$ if Σ is an embedding of Σ' and $\mathcal{D}$ and $\mathcal{D}'$ have the same interpretation for the symbols in Σ, the common part of their signatures. The algebra $\mathcal{D}'$ is also called an *enlargement* of $\mathcal{D}$. For a set of typed variables $\mathcal{X}$, a valuation v is a mapping $v : \mathcal{X} \to \mathcal{D}$. It can be extended in a unique way to a mapping from terms to $\mathcal{D}$.

[1]We use the term open function in analogy with *Open Logic Programming* [6] where a logic program may contain predicates which are not defined by program rules and are called *open predicates.*

A first-order Σ-*formula* is built from the logical connectives $\wedge, \vee, \neg, \leftarrow, \rightarrow, leftrightarrow$ and quantifiers over variables $\forall, \exists$ in the usual way. The existential and universal closures of a formula ϕ are denoted with $\tilde{\exists}\phi$ and $\tilde{\forall}\phi$. A Σ-*theory* is a set of closed Σ-formulae. A *model* of a Σ-theory $\mathcal{T}$ is a Σ-structure $\mathcal{D}$ such that all formulae of $\mathcal{T}$ evaluate to *true* under the interpretation provided by $\mathcal{D}$. A $\mathcal{D}$-*model* of a theory $\mathcal{T}$ is a model of $\mathcal{T}$ which is an enlargement of $\mathcal{D}$. We write $\mathcal{T}, \mathcal{D} \models \phi$ to denote that the formula ϕ is valid in all $\mathcal{D}$-models of $\mathcal{T}$.

3. CONSTRAINT DOMAINS WITH OPEN FUNCTIONS

The $CLP(\mathcal{C})$ framework [18] is parameterized by a particular constraint domain $\mathcal{C}$ which is a quadruple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ where Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ formulae and $\mathcal{T}$ is a first-order Σ-theory. The signature is assumed to contain the equality predicate $=$ and the disequality predicate $\neq$ for each sort s. The former is interpreted as identity in $\mathcal{D}$, the latter is its negation.

We consider an extension of a constraint domain $\mathcal{C}$ with a set of open functions Φ which will be denoted as $\mathcal{C}(\Phi)$. The first step is to extend the constraint signature Σ in the following way:

Definition 1. Let $\Sigma = \langle \mathcal{S}, \mathcal{F} \rangle$ be a signature and Φ a set of function symbols. Then $\Sigma(\Phi) = \langle \mathcal{S}, \mathcal{F} \cup \Phi \rangle$ is a *signature with open functions* Φ based on Σ.

It follows from this definition that Σ is embedded in $\Sigma(\Phi)$. Note that the arity of the open functions is defined over the set of sorts $\mathcal{S}$ from the base signature Σ. In the following, open functions will be denoted with the Greek letter ϕ.

Also, the structure $\mathcal{D}$ which gives the intended interpretation of the constraints needs to be extended in the presence of the open functions. In fact, such an extension will be a solution to a problem represented in our framework. More formally, we consider a $\Sigma(\Phi)$ structure $\mathcal{D}^\Phi$ which is an enlargement of $\mathcal{D}$. It gives an interpretation of the open functions for the set of carriers defined by $\mathcal{D}$.

We require the language $\mathcal{L}$ of the constraint domain $\mathcal{C}$ to be closed at least under variable renaming, conjunction and disjunction. Allowing a full first-order language requires to impose a restriction: universally quantified variables should not appear as arguments of open functions. The reason is that in this case we will be able to express arbitrary equational problems, hence the satisfiability test becomes undecidable. Decidability for arbitrary first-order formulae can only be ensured in special cases, e.g. when the constraint structure $\mathcal{D}$ has finite carriers for the domains of all open functions.

The first-order Σ-theory $\mathcal{T}$ is an axiomatization of some of the properties of $\mathcal{D}$. As any constraint includes the equality predicate $=$, every theory $\mathcal{T}$ contains the standard equality axioms for reflexivity, symmetry, transitivity and function and predicate substitution for the symbols in Σ. When the signature is extended with open functions we only need to add the function substitution axioms for the open functions:

$$\forall \tilde{x}, \tilde{y}.\ \tilde{x} = \tilde{y} \rightarrow \phi(\tilde{x}) = \phi(\tilde{y})$$

Let $\mathcal{T}^{\Phi}$ denote the theory $\mathcal{T}$ extended with the above axioms.

An important requirement of the theory $\mathcal{T}$ is *satisfaction completeness* [17] with respect to $\mathcal{L}$ - that is $\mathcal{D}$ is a model of $\mathcal{T}$ and for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \tilde{\exists} c$ or $\mathcal{T} \models \neg\tilde{\exists} c$. When working with open functions the first condition can be reformulated as: any enlargement $\mathcal{D}^{\Phi}$ of $\mathcal{D}$ is a model of $\mathcal{T}^{\Phi}$. The second condition does not hold in general (even if we use the theory $\mathcal{T}^{\Phi}$) because only some of the models of $\mathcal{T}^{\Phi}$ will be models of $\tilde{\exists} c$.

3.1 Constraint solving

Any constraint domain should support a test for satisfiability $\mathcal{D} \models \tilde{\exists} c$ which we call *constraint solving*. In practice, it is necessary not only to detect satisfiability of a given constraint but also to compute a $\mathcal{D}$-valuation v such that $\mathcal{D} \models v(c)$. We call such valuation a *$\mathcal{D}$-solution* to a constraint c and denote with $sol_{\mathcal{D}}(c)$ the set of all solutions: $sol_{\mathcal{D}}(c) = \{v | \mathcal{D} \models v(c)\}$. We say that two constraints c_1 and c_2 are *equivalent* if $sol_{\mathcal{D}}(c_1) = sol_{\mathcal{D}}(c_2)$ which can be expressed logically as $\mathcal{D} \models \tilde{\forall}(c_1 leftrightarrow c_2)$. Note that equivalence with respect to a structure $\mathcal{D}$ is implied by logical equivalence which holds for any structure. When working with open functions, equivalence of two constraints means that they have the same set of solutions for all enlargements $\mathcal{D}^{\Phi}$ of $\mathcal{D}$.

A standard approach for constraint solving is to transform a constraint c to a set of constraints in a *solved form* which satisfy the following properties:

- *Solvability* - a solved form has at least one solution.
- *Simplicity* - every solution can be easily obtained from a solved form.
- *Completeness* - every problem is equivalent to a finite disjunction of solved forms.

A well-known solved form in the domain of equality of finite trees is the *unification solved form* which has the form $\perp$ or $x_1 = t_1 \wedge \ldots \wedge x_n = t_n$ where the x_i are variables that occur only once.

A constraint solver can be implemented as a set of rewrite rules R. Rewriting a constraint c to a constraint c' is denoted with $c \mapsto c'$. A rule can be applied at any occurrence of a term matching the left hand side of a rewrite rule and we assume that the rewriting is done modulo the boolean properties (such as commutativity and associativity of conjunction and disjunction). A rule $c \mapsto c'$ is *correct* if c is equivalent to c'. A rewrite system is *terminating* when always a form is reached that cannot be further rewritten after a finite number of steps.

When extending a constraint domain with open functions, the interest is not only in the valuations under which a constraint is true, but, more importantly, in the enlargements $\mathcal{D}^{\Phi}$ of the structure $\mathcal{D}$ of the constraint domain used to obtain the valuations. A compact way to represent enlargements is by a conjunction of equations of the form $\phi(\tilde{s}) = t$ where $\tilde{s}$ and t are terms without open functions. We call such equations *open function equations*. Consequently, a constraint will have a form c, F where F contains only open function equations and c may contain arbitrary constraints. It is also natural to restrict the application of the rules of the rewrite system R to constraints not containing open functions. Now we define a solved form for constraints with open functions:

Definition 2. A constraint of the form c, F is in *open function solved form* if c is a constraint without open functions that is in solved form with respect to the original constraint domain, F is a conjunction of open function equations, and, for any $\mathcal{D}$-valuation v that is a solution of c there exists a structure $\mathcal{D}^{\Phi}$ that is a model of F.

This definition is partly syntactical (open functions must be isolated in the F part) and partly semantical (existence of an enlargement). In what follows we will develop rewrite rules which transform a constraint in a form that satisfies the conditions stated in the definition.

This definition meets the simplicity criterion because it implies that a solution for c (which is easily obtained because c is in solved form) is also a solution for c, F under any enlargement $\mathcal{D}^{\Phi}$ that interprets the open functions as $\mathcal{D}^{\Phi}(\phi(v(\tilde{s_i}))) = v(t_i)$ for all open function equations $\phi(\tilde{s_i}) = t_i \in F$. The solved form of c gives a compact representation for the (possibly infinite) set of valuations which satisfy c and the set of open function equations F gives a compact representation for the (possibly infinite) set of enlargements.

To obtain an open function solved form, a first step is to separate the open functions from the rest of the constraints by using the following normalization rule:

$$c[\phi(\tilde{s})], F \mapsto c[x], F \wedge \phi(\tilde{s}) = x \qquad \text{(N)}$$

where x is a new variable. Then the following proposition is easy to prove.

LEMMA 1. *The rule* (N) *is correct.*

After applying this rule a finite number of times, a problem can be put in a form c, F where c is a constraint without open functions and F is a conjunction of open function equations. We use this convention for c and F in the rest of this section.

Using the rule (N) is not enough for reaching a solved form. Indeed there can be a $\mathcal{D}$-valuation v such that $\mathcal{D} \models v(c)$ and that $\mathcal{D}$ cannot be extended in a model of F. This is the case when F contains a pair of open function equations $\phi(\tilde{s_i}) = t_i$ and $\phi(\tilde{s_j}) = t_j$ such that $v(\tilde{s_i}) = v(\tilde{s_j})$ but $v(t_i) \neq v(t_j)$. To solve this problem, we introduce another rule:

$$c, F \mapsto c \wedge (s_1 \neq t_1 \vee \ldots \vee s_n \neq t_n \vee r_1 = r_2), F \qquad (FS)$$

where $\phi(s_1, \ldots, s_n) = r_1, \phi(t_1, \ldots, t_n) = r_2 \in F$. The rule is applied exactly once for each distinct pair of open function equations in F. Note that the disjunction in the right-hand side is equivalent to the implication

$$s_1 = t_1 \wedge \ldots \wedge s_n = t_n \rightarrow r_1 = r_2$$

If $true, F \mapsto^*_{FS} c, F$ where $\mapsto^*_{FS}$ is the closure of the rule FS then we denote with $FS(F)$ the formula c. Note that it does not contain open functions.

Lemma 2. *The rule (FS) is correct.*

Proof. We prove that the rule preserves the logical equivalence. First, we add to F the function substitution axiom for the equality predicate $\tilde{x} = \tilde{y} \rightarrow \phi(\tilde{x}) = \phi(\tilde{y})$ which is valid in any structure and apply the substitution $\{\tilde{x}/\tilde{s}, \tilde{y}/\tilde{t}\}$. We obtain the formula $\tilde{s} = \tilde{t} \rightarrow \phi(\tilde{s}) = \phi(\tilde{t})$ which is equivalent to $\tilde{s} = \tilde{t} \rightarrow r_1 = r_2$ by using the symmetry and transitivity rule and the equations $\phi(\tilde{s}) = r_1 \wedge \phi(\tilde{t}) = r_2$. The other direction is trivial. $\square$

Lemma 3. *If v is $\mathcal{D}$-valuation which is a solution of $FS(F)$ then there exists an enlargement $\mathcal{D}^\Phi$ of $\mathcal{D}$ such that $\mathcal{D}^\Phi \models v(F)$.*

Proof. Let v be a $\mathcal{D}$-valuation such that $\mathcal{D} \models v(FS(F))$. Then, we can construct a structure $\mathcal{D}^\Phi$ which satisfies F by induction on the number of equations in F. Suppose that $\mathcal{D}^\Phi_{i-1}$ is a partial interpretation which is a model of the first $i-1$ equations from F and consider the ith equation $\phi(\tilde{s_i}) = t_i$. If $\mathcal{D}^\Phi_{i-1}$ already defines the value of ϕ for $v(\tilde{s_i})$ then there exists some equation $\phi(\tilde{s_j}) = t_j$ with $j < i$ such that $v(\tilde{s_j}) = v(\tilde{s_i})$. But then since v is a solution to $FS(F)$ we have that $v(t_j) = v(t_i)$. Otherwise we extend $\mathcal{D}^\Phi_{i-1}$ to $\mathcal{D}^\Phi_i$ by defining $\mathcal{D}^\Phi_i(\phi(v(\tilde{s_i}))) = v(t_i)$. $\square$

To illustrate the application of the two rules we consider a simple example adapted from [15].

Example 1. *Let the constraint domain be the algebra of finite trees and consider the following conjunction of open function equations:*

$$\phi(a, 1) = X \wedge \phi(X, 1) = b(Y)$$

We will use some simple rules for rewriting equations between terms which can be found in e.g. [5]:

$$true, \underline{\phi(a,1) = X} \wedge \underline{\phi(X,1) = b(Y)} \mapsto_{FS}$$
$$(X \neq a \vee \underline{1 \neq 1} \vee X = b(Y)), \phi(a,1) = X \wedge \phi(X,1) = b(Y) \mapsto$$
$$(\underline{X \neq a} \vee X = b(Y)), \phi(a,1) = X \wedge \phi(X,1) = b(Y) \mapsto_{Replace}$$
$$(X \neq a \vee \underline{a = b(Y)}), \phi(a,1) = X \wedge \phi(X,1) = b(Y) \mapsto_{Clash}$$
$$X \neq a, \phi(a,1) = X \wedge \phi(X,1) = b(Y)$$

Theorem 1. *Let R be a correct and terminating set of rules which transform any constraint without open functions to an equivalent finite disjunction of constraints in solved form. Then $R \cup \{N, FS\}$ is a correct and terminating set of rules for constraints with open functions which transform any constraint c to an equivalent finite disjunction of constraints that are in open function solved form.*

Proof. The correctness follows from lemmas 1 and 2 and the correctness of R. For what concerns termination it follows from the definitions of the rules N and FS that they derive in a finite number of steps a system of the form c, F such that c does not contain open function symbols and F cannot be further reduced by N and FS. As R is terminating and can only select constraints without open functions it reduces in a finite number of steps to c', F' with c' the solved form of c which cannot be further reduced by N and FS. Hence, by lemma 3, it is in open function solved form. This shows the existence of a terminating derivation. Intertwining the application of R with the application of N and FS cannot endanger termination as R cannot select the equations that N and FS can select. $\square$

This result shows that we can reduce the problem of finding a solution to a constraint with open functions to solving a constraint without open functions. A set of rules for the domains of finite, rational and infinite trees which satisfy the conditions of the above theorem can be found in [5].

3.2 Finite Domain Constraints

Given a constraint c with n occurrences of open function symbols then the rule N can be applied at most n times and consequently the rule FS can be applied $n!$ times. In many practical CLP applications it is preferable to have a constraint solver with smaller complexity even if it is not a complete solver. A typical example is finite domain constraint solving where the search is divided in two parts. First, a proof procedures constructs a set of constraints using an incomplete but efficient test for satisfiability. Typically this is some algorithm for guaranteeing local consistency of the set of constraints by reducing the domains of the variables. Then follows an *enumeration* phase where the CLP variables are assigned values and tested for consistency. By taking into account these properties of finite constraint domains, an incomplete algorithm for solving open functions with smaller complexity can be devised. First, as the domain of the open functions is finite, we can construct a conjunction of open function equations which represents the whole interpretation for a given open function ϕ. Let $T_{\mathcal{D}}(\phi)$ denote such conjunction:

$$T_{\mathcal{D}}(\phi) \equiv \phi(\tilde{d_1}) = x_1 \wedge \ldots \wedge \phi(\tilde{d_n}) = x_n$$

for a given open function ϕ where $\tilde{d_i}$ are all possible tuples of ground terms respecting the sort of ϕ and x_i are new variables. Then, all occurrences of open functions in a constraint c will be normalized in a similar way as before, however the resulting open function equations will be treated as normal constraints.

More formally, we represent the current state of the computation as a pair c, T where c is a conjunction of constraints and T is a conjunction of open function equations which will describe the whole interpretation of each open function $\phi \in \Phi$. The rule for normalizing open functions has the following form:

$$c[\phi(\tilde{s})], T \mapsto c[x] \wedge \phi(\tilde{s}) = x, T \qquad \text{(N1)}$$

if $T_{\mathcal{D}}(\phi) \in T$ and

$$c[\phi(\tilde{s})], T \mapsto c[x] \wedge \phi(\tilde{s}) = x, T \wedge T_{\mathcal{D}}(\phi) \qquad \text{(N1)}$$

if $T_{\mathcal{D}}(\phi) \notin T$. We also assume that the rule is not applied for an open function ϕ which appears at the principal position of an open function equation $\phi(\tilde{s}) = t$. The idea of the rule $N1$ is that for each open function ϕ which is normalized, a conjunction of equations describing its interpretation is added to T.

Because of the properties of finite domain constraints, discussed before, the arguments of the open functions will always become ground at some time (at least during an enumeration step). Hence, the following rule can be used to replace a ground instance of an open function ϕ with the variable representing its interpretation:

$$c[\phi(\tilde{d_i})], T \wedge \phi(\tilde{d_i}) = z_i \mapsto c[z_i], T \wedge \phi(\tilde{d_i}) = z_i \qquad \text{(S)}$$

THEOREM 2. *Let $\mathcal{C}(\Phi)$ be a constraint domain with open functions such that the carriers of all open functions $\phi \in \Phi$ are finite and let R be a correct and terminating set of rules which transform any constraint without open functions to an equivalent finite disjunction of constraints in solved form. Then $R \cup \{N1, S\}$ is correct and terminating set of rules for constraints with open functions which transform any constraint c to an equivalent finite disjunction of constraints that are in open function solved form.*

Finite domains constraint solving is implemented by associating with each variable x, a domain of possible values D_x and then reducing these domains until the set of constraints becomes locally consistent [33]. For open function equations $\phi(\tilde{x}) = y$ it is also possible to give rules for reducing the domains of $\tilde{x}$ and y. Suppose the current state is

$$c \wedge \phi(\tilde{x}) = y, T \wedge \phi(\tilde{d_1}) = z_1 \wedge \ldots \wedge \phi(\tilde{d_n}) = z_n$$

with selected constraint $\phi(\tilde{x}) = y$. The following rule can be used to reduce the domain of y:

$$D_y = D_y \cap \bigcup_{\tilde{d_i} \in D_{\tilde{x}}} D_{z_i} \qquad \text{(P1)}$$

Here with $\tilde{d_i} \in D_{\tilde{x}}$ we have denoted all tuples $\langle d_{i1}, \ldots, d_{im} \rangle$ of domain elements such that $d_{i1} \in D_{x_1}, \ldots, d_{im} \in D_{x_m}$. The second propagation rule is used to reduce the domains of the arguments $\tilde{x}$ of the open function. It will be computed in two steps. First we compute a relation R with the possible tuples $\tilde{d}$ for $\tilde{x}$:

$$R = \{\tilde{d_i} \mid D_{z_i} \cap D_y \neq \emptyset\}$$

Then the domain of each individual variable x_i is reduced to the projection of R on the ith argument:

$$D_{x_i} = D_{x_i} \cap R_i \qquad \text{(P2)}$$

Most of the current finite domain CLP systems include a constraint $element(x, [z_1, \ldots, z_n], y)$ [10] which is true if y is equal to the xth element of the list, z_x. If ϕ is an open function of one argument with domain $1, \ldots, n$ then a constraint

$$c \wedge \phi(x) = y, T \wedge \phi(1) = z_1 \wedge \ldots \wedge \phi(n) = z_n$$

can be directly mapped to

$$c \wedge element(x, [z_1, \ldots, z_n], y), \\ T \wedge \phi(1) = z_1 \wedge \ldots \wedge \phi(n) = z_n$$

In fact the above two rules for solving open function equations are generalizations of rules for solving the *element*/3 constraint.

4. LANGUAGE AND SEMANTICS

Using open functions is not enough to achieve a good declarative representation for most of the problems. Therefore, we also introduce the concept of *integrity constraints* which express properties of the intended solutions. Formally, we define a problem with open functions to be a triple $\langle \mathcal{C}(\Phi), \mathcal{P}, \mathcal{IC} \rangle$ where $\mathcal{C}(\Phi)$ is a constraint domain $\mathcal{C}$ extended with open functions Φ, $\mathcal{P}$ is a constraint logic program and $\mathcal{IC}$ is a set of integrity constraints. The program $\mathcal{P}$ is a set of clauses of the form

$$A \;:-\; c\,,\; L_1, \ldots, L_n$$

with A an atom, c a constraint (with open functions), and L_j literals. As in Prolog, program clauses use the symbol `:-` to separate head from body. In general, the integrity constraints can be arbitrary first-order $\Sigma(\Phi)$-formulae. However, we assume that that they are in a clausal form

$$A_1 \vee \ldots \vee A_m \;\leftarrow c\,,\; L_1, \ldots, L_n$$

with the head a disjunction of atoms A_i and the body a conjunction of a constraint c and of literals L_j[2]. Unlike the situation in standard logic programming, where a complex transformation, like Lloyd-Topor, should be used to put a set of first-order formulae in clausal form, here we can just put a formula in prenex normal form, skolemize it and add all skolem functions as new open functions. Also, if one is interested in a query $\leftarrow q(\tilde{X})$ it can be put as an integrity constraint $q(\tilde{a}) \leftarrow true$ where $\tilde{a}$ are new open constants. An interpretation of these constants in a solution $\mathcal{D}^{\Phi}$ will correspond to an answer substitution for the original variables from the query.

As an example, we give the specification of the n-queens problem for the constraint domain of finite domain integer arithmetic.

EXAMPLE 2. *N-queens*

```
% Declarations
constant size == 8.
domain row == 1..size.
domain column == 1..size.
open_function pos(row):column.
```

[2]To stress that the formula is an integrity constraint, the propositional constant *false* is used as left hand side when $m = 0$ and *true* is used as right hand side when $n = 0$.

```
% Program
attack(R1,R2) :- R1 < R2, pos(R1) = pos(R2).
attack(R1,R2) :- R1 < R2, R2-R1 = abs(pos(R2)-pos(R1)).

% Integrity Constraints
false <- attack(R1,R2).
```

The program consists of three parts. The first part contains declarations and starts with defining a constant which is the parameter of the problem. Next, the sorts row and column are introduced together with the size of the domains. Finally, the open function pos $: row \rightarrow column$ *is declared. The program part defines a single relation* $attack/2$ *which is true when the queen on row* $R1$ *attacks a queen on a higher row* $R2$ *(this avoids the overhead of performing symmetric tests). The constraint part defines the single constraint that the* $attack/2$ *relation must be empty.*

To give a semantics to a problem $\langle \mathcal{C}(\Phi), \mathcal{P}, \mathcal{IC}\rangle$ we revert to the different semantics developed for standard logic programming.

Definition 3. Let $\mathcal{D}^\Phi$ be a $\Sigma(\Phi)$-structure which is an enlargement of the constraint structure $\mathcal{D}$. We define the grounding of the program $\mathcal{P}$ with respect to $\mathcal{D}^\Phi$ as

$$\mathcal{P}_G(\mathcal{D}^\Phi) = \{v(A \;:-\; B) \mid (A \;:-\; c\,,\, B) \in \mathcal{P} \text{ and } \mathcal{D}^\Phi \models v(c) \text{ for a } \mathcal{D}\text{-valuation } v\}$$

Then $\mathcal{D}^\Phi$ is a solution to a problem $\langle \mathcal{C}(\Phi), \mathcal{P}, \mathcal{IC}\rangle$ if the integrity constraints $\mathcal{IC}$ evaluate to true in all canonical $\mathcal{D}^\Phi$-model(s) of $\mathcal{P}_G(\mathcal{D}^\Phi)$.

Depending on the knowledge representation and computational requirements of the particular problem, different semantics can be considered for the canonical model(s). In the rest of the paper we will consider three valued semantics of the Clark's completion of the program [3] and the well-founded semantics [34].

5. PROOF PROCEDURE

The Definition 3 of a solution to a problem is not very useful in practice as it only gives a method for testing if an interpretation of the open functions $\mathcal{D}^\Phi$ is a solution. By using three-valued models of the Clark's completion of the program [12] as a semantics for our framework, a direct representation of $\mathcal{D}^\Phi$ in terms of a constraint c can be obtained. For a given program $\mathcal{P}$ let $\mathcal{P}^*$ be the completion of the program [3] (without the Clark's equality theory). First we put the integrity constraints $\mathcal{IC}$ as part of the program. Let ic be a new propositional constant, then a rule $ic :- c, L_1, \ldots, L_n, not(A_1), \ldots, not(A_m)$ is created for each integrity constraint $A_1 \vee \ldots \vee A_m \;\leftarrow c\,,\, L_1, \ldots, L_n \in \mathcal{IC}$. Let $\mathcal{P}_{\mathcal{IC}}$ denotes the resulting program. Then it is easy to see that

$$\mathcal{P}^*_{\mathcal{IC}} \models_3 not(ic) leftrightarrow \mathcal{IC} \qquad (1)$$

The proof procedure is presented as a variant of the Apt-Doets SLDNF proof procedure [1] generalized for CLP and further enhanced with elements from constructive negation [32]. The SLDNF procedure builds a set of trees (a forest), one being the main tree. Nodes in the tree are goals of the form $\leftarrow c \mid L_1, \ldots, L_m$ where c is a constraint and L_i are literals. With $m = 0$ and c a satisfiable constraint, the node is a *success* node and is labeled as such. A leaf node which cannot be extended with any descendant and is not a success node is labeled as *failure* node. A tree is *complete* when no node can be extended with new descendants and all leaf nodes are labeled with success or failure. When processing a leaf node, the computation rule selects a literal. For simplicity of presentation, we assume always the leftmost literal is selected.

- Initialization. Given a conjunction of literals $L_1, \ldots, L_n$, the main tree is initialized with root node $\leftarrow true \mid L_1, \ldots, L_n$.
- Extension of leaf node with positive literal selected. Let the node be $\leftarrow c \mid p(\tilde{X}), L_2, \ldots, L_m$. For every rule (renamed apart) $p(\tilde{Y}) \;:-\; c_0\,,\, B_1, \ldots, B_k$ such that $c \wedge c_0 \wedge \tilde{X} = \tilde{Y}$ is satisfiable, add an immediate descendant with node $\leftarrow c \wedge c_0 \wedge \tilde{X} = \tilde{Y} \mid B_1, \ldots, B_k, L_2, \ldots, L_m$. If the constraint is not satisfiable for any of the rules defining p/n then label the node with failure.
- Extension of leaf node with negative literal selected. Let the node be $\leftarrow c \mid not(p(\tilde{X})), L_2, \ldots, L_m$. Let c' be some constraint which is implied by the projection of c on $\tilde{X}$.
  - There is no tree with root node (a renaming of) $\leftarrow c' \mid p(\tilde{X})$: create such a tree (this tree is subsidiary to the leaf node).
  - The subsidiary tree is complete and has success nodes labeled $\leftarrow c' \wedge c_1 \mid \Box, \ldots \leftarrow c' \wedge c_n \mid \Box$. Let $C = c \wedge \neg\exists \tilde{Y_1}\, c_1 \wedge \ldots \wedge \neg\exists \tilde{Y_n}\, c_n$ where $\tilde{Y_i}$ are all the variables from c_i and $\tilde{X}$ which do not appear in $c, L_2, \ldots, L_m$. If C is satisfiable then add an immediate descendant with node $\leftarrow C \mid L_2, \ldots, L_m$, else label the node with failure[3].

Note that a derivation is nonterminating when a negative literal is selected and the subsidiary tree cannot be completed. The construction $\neg\exists\tilde{Y}$ in the rule for negation introduces universally quantified variables. As we do not allow in our language universally quantified variables as arguments of open functions, the computation *flounders* in such a situation. This limitation can be removed when the domains of the open functions are finite (all examples from the paper fall in this category).

The procedure delegates the handling of the open terms to the constraint solver which is discussed in section 3.1. The ability to weaken the constraint when creating the root of a subsidiary tree (e.g. by dropping some of the constraints involving open functions) is a way to replace a global satisfiability check by a less expensive local one. Consistency

[3]Note that failure can sometimes be detected without computing the complete subsidiary tree, i.e. as soon as $c \wedge \neg\exists\tilde{Y_1}\, c_1 \wedge \ldots \wedge \neg\exists\tilde{Y_i}\, c_i$ is false for some i (e.g., this is the case when $c_i = true$ for some i).

with the parent is not checked until the subsidiary tree is complete. However, this may cause nontermination of the subsidiary tree and may compromise the completeness of the proof procedure. While this can increase the size of the forest, it will typically be compensated by the reduced cost of the satisfiability checks.

Note that this proof procedure can be seen as an instance of the constructive negation framework for CLP programs [32]:

THEOREM 3 (SOUNDNESS). *Let $\mathcal{P}$ be a logic program over a constraint domain $\mathcal{C}(\Phi)$. If the goal $\leftarrow true \mid G$ has a success node $\leftarrow c \mid \Box$ of the main tree then $\mathcal{P}^*, \mathcal{D} \models_3 \tilde{\forall}(c \rightarrow G)$.*

THEOREM 4. *Let $\langle \mathcal{C}(\Phi), \mathcal{P}, \mathcal{IC} \rangle$ be a problem. If $\leftarrow c|\Box$ is a success node of the main tree for the goal $\leftarrow true \mid not(ic)$ and $\mathcal{D}^\Phi$ an enlargement of the structure $\mathcal{D}$ such that $\mathcal{D}^\Phi \models \tilde{\exists} c$ then $\mathcal{P}^*, \mathcal{D}^\Phi \models_3 \mathcal{IC}$.*

PROOF. If $\leftarrow c \mid \Box$ is a success node of the main tree for the goal $\leftarrow true \mid not(ic)$ for the program $\mathcal{P}_{\mathcal{IC}}$ then from Theorem 3 follows that $\mathcal{P}^*_{\mathcal{IC}}, \mathcal{D} \models_3 (\tilde{\exists} c) \rightarrow not(ic)$. Now for any $\Sigma(\Phi)$ enlargement $\mathcal{D}^\Phi$ of $\mathcal{D}$ such that $\mathcal{D}^\Phi \models \tilde{\exists} c$ follows that

$$\mathcal{P}^*_{\mathcal{IC}}, \mathcal{D}^\Phi \models_3 not(ic) \qquad (2)$$

and from (1) we have that $\mathcal{P}^*_{\mathcal{IC}}, \mathcal{D}^\Phi \models_3 \mathcal{IC}$. Now we will show that $\mathcal{P}^*, \mathcal{D}^\Phi \models_3 \mathcal{IC}$. Suppose that there exist a three valued $\mathcal{D}$-model M such that $M, \mathcal{D}^\Phi \models \mathcal{P}^*$ and $M, \mathcal{D}^\Phi \not\models \mathcal{IC}$. This means that there is an integrity constraint $A_1 \vee \ldots \vee A_m \leftarrow c\,,\, L_1, \ldots, L_n \in \mathcal{IC}$ and $\mathcal{D}$-valuation v such that $\mathcal{D}^\Phi \models v(c)$, $M \models v(L_1), \ldots, v(L_n)$ and $M \not\models v(A_1) \vee \ldots \vee v(A_m)$ or equivalently $M \models not(v(A_1)), \ldots, not(v(A_m))$. With $M' = M \cup \{ic\}$ one obtains a model of $\mathcal{P}^*_{\mathcal{IC}}$ because $icleftrightarrow \ldots \vee \tilde{\exists} X L_1, \ldots, L_n, not(B_1), \ldots, not(B_m) \vee \ldots$. Hence $M', \mathcal{D}^\Phi \models \mathcal{P}^*_{\mathcal{IC}}$ which contradicts with (2). $\Box$

Soundness also holds for stronger semantics such as the well-founded semantics. Completeness does not hold; however a weaker result can be stated:

THEOREM 5 (WEAK COMPLETENESS). *If there exists an enlargement $\mathcal{D}^\Phi$ of $\mathcal{D}$ such that $\mathcal{D}^\Phi, \mathcal{P}^* \models_3 \mathcal{IC}$, then the derivation starting in $\leftarrow true \mid not(ic)$ does not fail.*

5.1 Tabulation

When adhering to the well-founded semantics [34] or the principle of inductive definitions [7] for the semantics of $\mathcal{P}_G(\mathcal{D}^\Phi)$, tabulation [2, 31] is sound, can improve the termination properties and reduces the number of redundant evaluations of subgoals. While weaker than SLG-resolution [2], adding the rule below for tabled predicates to the procedure is sound and weakly complete under the well-founded semantics.

- Extension of leaf node with positive literal selected. Let the node be $\leftarrow c_0 \mid p(\tilde{X}), L_2, \ldots, L_m$. Let c' be the projection of c_0 on $\tilde{X}$.
  - If there is no tree with root node (a renaming of) $\leftarrow c' \mid p(\tilde{X})$ then create such a tree (this tree is subsidiary to the leaf node).
  - If the subsidiary tree has a success node $\leftarrow c' \wedge c | \Box$ then use $p(\tilde{X})\ :-\ c' \wedge c\,,\ true$ as a rule to compute a new descendant for the node $\leftarrow c_0 \mid p(\tilde{X}), L_2, \ldots, L_m$.
  - If the subsidiary tree is complete[4] and has no success nodes then label the node $\leftarrow c_0 | p(\tilde{X}), L_2, \ldots, L_m$ with failure.

6. ADVANCED EXAMPLES ILLUSTRATING TABLING AND NEGATION

In this section we present a declarative specification of some problems which are frequently used to illustrate the declarative merits of stable logic programming.

EXAMPLE 3. *Hamiltonian path.*

An example requiring tabulation is the problem of finding a closed path in a graph which visits each node exactly once.

```
domain node == 1..n.
open_function ham(node):node.

% PROGRAM
edge(1,2).
...
% connects/2: nodes connected by the hamiltonian path
connects(X,Y) :- ham(X)=Y.
connects(X,Y) :- ham(X)=Z, connects(Z,Y).

% CONSTRAINTS
% path must be following the given edges
edge(X,Y) <- ham(X)=Y.
% each node has at most one incoming path
X=Y <- ham(X)=Z, ham(Y)=Z.
% all nodes must be connected with node 1
connects(1,X) <- node(X).
```

The hamiltonian path is given by an open function `ham/1` *which maps each node to the next one in the path. The definition of* `connects/2` *is recursive and the evaluation of the integrity constraint containing* `connects/2` *literals is non-terminating under SLDNF. Tabling the* `connects/2` *predicate ensures termination. We have a prototype implementation with tabling incorporated. It is able to solve this problem. Other abductive systems such as SLDNFA [9] are trapped in a loop by the recursive definition of connects. Note that the answers for the* `connects/2` *predicates will be constrained facts, hence constraint normalization and subsumption testing is essential to ensure that only a finite number of distinct answers is obtained.*

[4]The completeness check becomes more involved as new branches can sprout from non-leaf nodes. Hence one needs to be sure this cannot happen before a tree can be considered as complete.

EXAMPLE 4. *Planning.*

The following program skeleton shows a solution for planning in a simple blocks world with one robot arm. Defining `clear/2` *(whether a block is clear at time t) in terms of* `on/3` *(which expresses that a block is on a location (another block or the table) at a time t), we need only one basic property (*`on/3`*) and we can avoid the meta-level of a* `holds/2` *predicate as e.g. in [11].*

```
constant maxstep == ....
% final state reached at time maxstep
domain step == 0..maxstep-1.
domain block == b1 | b2 ...
domain table == table.
domain location == block + table. %union
domain action == move(block,location,location) | skip.
open_function plan(step):action.

% PROGRAM
on(B,L,0) :- initially_on(B,L)).
on(B,L,T0+1) :- plan(T0)=A, initiates(A,on(B,L)).
on(B,L,T0+1) :-
    on(B,L,T0), plan(T0)=A, not(terminates(A,on(B,L)).
initiates(move(B,S,D),on(B,D)).
terminates(move(B,S,D),on(B,S)).
clear(B,T) :- block(B),not(covered(B,T)).
clear(table,T).
covered(B,T) :- on(B1,B,T).
% facts describing initial configuration
initially_on(...).

% CONSTRAINTS
on(B,S,T) <- plan(T)=move(B,S,D).
clear(B,T) <- plan(T)=move(B,S,D).
clear(D,T) <- plan(T)=move(B,S,D).
% description of final configuration
on(..,..,maxstep) <- true.
```

The open function `plan/1` *describes the plan as actions being performed in successive steps (time points). Its argument, the domain* `step`*, describes the time points at which actions are allowed. Its range, the domain* `action`*, describes the set of possible actions. Beside the* `skip` *action, it includes also a set of* `move` *actions which are compound terms. The domains* `block`*,* `table` *and* `location` *are auxiliary domains used to define the* `action` *domain. The* `on/3` *predicate is defined by three clauses. The first one gives the initial location of the blocks (at time 0) which are given by a set of problem specific* `initially_on/2` *facts (in a more general problem,there could be another open function mapping blocks to their initial position). The next two clauses give the frame axioms: a block is at a location in the time point following an action if either that action initiates the property or if it was there before and the action does not terminate the property. The first three constraints express the conditions a valid move has to satisfy: the moved block has to be at the origin of the move, it has to be clear and the destination has to be clear. The problem specific constraints describe the target configuration to be reached at time* `maxstep`*.*

7. RELATED WORK

The idea to extend the semantics of constraint logic programming to a set of possible models goes back to [16]. More recent, Hickey [15] studied the extension of CLP with open functions. He considers a constraint domain $Fun(\mathcal{C})$ which is obtained by extending a constraint domain $\mathcal{C}$ with a new binary open function ϕ. He also gives an algorithm for solving constraints with open function in the domain of equations and disequations of finite trees. His procedure is based on the work of Colmerauer [4] on solving disequations in Prolog II. Although in [15] are considered only signatures with infinite many function symbols, the results are also applicable to the case with finite number of function symbols and infinite number of terms. In our view, one disadvantage of his approach is that the solved form defined there does not satisfy the simplicity requirement. To solve constraints with open functions, he uses the following merging rule which is an instance of the rule FS:

$$\phi(\tilde{s}) = t_1, \phi(\tilde{s}) = t_2 \mapsto \phi(\tilde{s}) = t_1, t_1 = t_2 \qquad (3)$$

With this rule, it is possible to have a valuation v which is a solution for a constraint c but for which no enlargement $\mathcal{D}^\Phi$ exists such that $\mathcal{D}^\Phi \models v(c, F)$. Consider the problem from Example 1 of solving the constraint

$$true, \phi(a,1) = X \wedge \phi(X,1) = b(Y)$$

It is in a solved form by the definition of [15], however any valuation v such that $v(X) = a$ cannot be a solution.

Also, the programming language in which Hickey embeds open functions retains the typical procedural programming style of CLP that follows from the concept of computing an answer for the variables in a query. The main originality of our work is that we embed open functions in a language supporting a much more declarative programming style.

Although we have presented open functions as an extension to a CLP-language, one could also consider it as a restricted form of abduction. In fact, there is a fairly simple translation to an abductive program: for each open function ϕ with sort $\tilde{w} \to s$ introduce an abductive predicate p_ϕ with sort $\tilde{w}, s$. Then replace every occurrence $\phi(\tilde{t})$ of an open function by a fresh variable X and add the abductive call $p_\phi(\tilde{t}, X)$ to the body of the clause or integrity constraint. Finally, add the following rule and integrity constraints which restrict the interpretation of the predicate p_ϕ to encode a function [30]:

$$exists\_p_\phi(\tilde{X}) :- p_\phi(\tilde{X}, Y).$$

$$exists\_p_\phi(\tilde{X}) \leftarrow true.$$
$$\tilde{Y} = \tilde{Z} \leftarrow p_\phi(\tilde{X}, Y), p_\phi(\tilde{X}, Z).$$

This shows that one could use abductive systems as an implementation of a language enriched with open functions. In fact, one could go further, using the known relationship between abduction an stable models [20], it looks feasible to further translate programs (at least if all other functors are constants) into input for the **smodels** system [29, 28]. However we believe a more promising path towards a high performance implementation is by a direct implementation which extends the solver of a CLP language with the machinery to handle open functions.

8. CONCLUSION

Constraint logic programming is a very powerful paradigm but has several drawbacks from point of view of knowledge representation. Solutions to problems have to be encoded in a data structure; this leads to a procedural programming style and a representation of the problem specific knowledge which is less direct and hence less declarative than one would wish. Recently some new paradigms have been proposed, notably a logic programming style where solutions are stable models of logic programs [27, 25, 26, 28] and abductive logic programming [19, 21, 22] where solutions are models of certain predicates. In both cases, solutions are relations which can be represented as tables. It turns out that these paradigms allow much more declarative representations of various problems within the scope of finite domain constraint logic programming. However **smodels**, one of the most advanced implementations supporting the stable logic programming paradigm is far from achieving a performance comparable to finite domain CLP systems; the same holds for abductive systems which have quite complex inference rules and are implemented as meta-interpreters on top of a CLP-system [30].

In this paper we have explored an alternative approach which consists of enriching the constraint domain of a constraint logic language with open functions. Also Hickey did this [15], however in addition we introduced a paradigm shift. We dropped the notion of query and computing answers for it. Instead, answers are provided by the interpretation of the open functions. To constrain them to solutions of the problem at hand, the program is augmented with integrity constraints. There is a lot of similarity between the functions which are the answers in our approach and the relations which are the answers in stable logic programming and abductive logic programming. The examples throughout the paper illustrate that our approach leads to elegant declarative problem specifications. It is fair to say that they are comparable to their counterparts in abductive and stable logic programming. We have analysed constraint solving with open functions and did substantially extend the work of Hickey [15]. We believe that our results are a sound basis for extending constraint solvers with the capability for handling functions with open interpretation. This offers interesting perspectives towards the realisation of systems which combine declarativity of representation with good performance.

9. ACKNOWLEDGMENTS

This first author is supported by the GOA project LP+. The second author is supported by FWO-Vlaanderen. The authors are grateful to Marc Denecker for his early collaboration on this topic and for many useful comments and discussions.

10. REFERENCES

[1] K. R. Apt and K. Doets. A new definition of SLDNF-resolution. *Journal of Logic Programming*, 18(2):177–190, Feb. 1994.

[2] W. Chen and D. Warren. Tabled evaluation with delaying for general logic programs. *Journal of ACM*, 43(1):20–74, 1996.

[3] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[4] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, Japan, Nov. 1984. OHMSHA Ltd. Tokyo and North-Holland.

[5] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7:371–425, 1989.

[6] M. Denecker. A terminological interpretation of (abductive) logic programming. In V. W. Marek and A. Nerode, editors, *Logic Programming and Nonmonotonic Reasoning, Proceedings*, volume 928 of *LNAI*, pages 15–28, Lexington, KY, USA, 1995. Springer.

[7] M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. F. del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence, European Workshop, JELIA*, volume 1489 of *Lecture Notes in Computer Science*, pages 1–16, Dagstuhl, Germany, Oct. 1998. Springer.

[8] M. Denecker and D. De Schreye. Terms in logic programs: a problem with their semantics and its effect on the programming methodology. *The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 7(3-4):363–383, 1990.

[9] M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, Feb. 1998.

[10] M. Dincbas, H. Simonis, and P. V. Hentenryck. Solving a cutting-stock problem in constraint logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 42–58, Seattle, Washington, Aug. 1988. MIT Press.

[11] K. Eshghi. Abductive planning with event calculus. In R. Kowalski and K. Bowen, editors, *Proc. of the International Conference on Logic Programming*, pages 562–579. The MIT press, 1988.

[12] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[13] T. F. Fung and R. A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, Nov. 1997.

[14] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming, Proc. Fifth Int. Conf. and Symp. (IJCSLP'88*, pages 1070–1080. MIT Press, 1988.

[15] T. J. Hickey. Functional constraints in CLP languages. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 355–381. MIT Press, 1993.

[16] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Oct. 1988.

[17] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc. of 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, Jan. 1987.

[18] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic programming*, 19-20:503–581, 1994.

[19] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Programming 5*, pages 235–324. Oxford University Press, 1998.

[20] A. C. Kakas and P. Mancarella. Generalized stable models: A semantics for abduction. In *Proceedings of the 9th ECAI*, pages 385–391, Stockholm, Sweden, Aug. 1990.

[21] A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 399–413. Tokyo, Japan, MIT Press, 1995.

[22] A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *Journal of Logic programming*, 44(1–3):129–177, 2000. Special issue Abductive Logic Programming.

[23] A. C. Kakas and C. Mourlas. ACLP: Flexible solutions to complex problems. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 387–398, Berlin, July 28–31 1997. Springer.

[24] H. A. Kautz and B. Selman. Planning as satisfiability. In B. Neumann, editor, *Proc. ECAI'92*, pages 359–363, Vienna, Austria, Aug. 1992. John Wiley and Sons.

[25] V. Lifschitz. Action languages, answer sets, and planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages 357–373. Springer-Verlag, 1999.

[26] V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Logic Programming, Proc. 1999 Int. Conf. on Logic Programming (ICLP'99)*, pages 23–37. MIT Press, 1999.

[27] V. M. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In K. Apt, V. W. Marek, M. Truszczyński, and D. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages 375–398. Springer-Verlag, 1999.

[28] 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.

[29] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Logic Programming, Proc. 1996 Joint Int. Conf. and Symp. on Logic Programming (JICLP'96)*, pages 289–303. MIT Press, 1996.

[30] N. Pelov, E. De Mot, and M. Bruynooghe. A comparison of logic programming approaches for representation and solving of constraint satisfaction problems. In M. Denecker, A. Kakas, and F. Toni, editors, *8th International Workshop on Non-Monotonic Reasoning, Special Session on Abduction*, Breckenridge, Colorado, USA, Apr. 2000.

[31] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of 1994 ACM SIGMOD*. ACM Press, 1994.

[32] P. J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 328–339, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[33] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, 1993.

[34] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[35] M. Wirsing. Algebraic specification. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Volume B: Formal Models and Semantics, pages 677–788. Elsevier and MIT Press, 1990.