Generalizing Finite Domain Constraint Solving

Federico Bergenti, Alessandro Dal Palù, and Gianfranco Rossi

Dipartimento di Matematica
Università degli Studi di Parma
Viale G. P. Usberti, 53/A
43100 Parma, Italy
{federico.bergenti,alessandro.dalpalu,gianfranco.rossi}@unipr.it

Abstract. This paper summarizes a constraint solving technique that can be used to reason effectively in the scope of a constraint language that supersedes common finite domain languages available in the literature. The first part of this paper motivates the presented work and introduces the constraint language, namely *Hereditarily Finite Sets* (*HFS*) language. Then, the proposed constraint solver is detailed in terms of a set of rewrite rules which exploit finite domain reasoning within the HFS language. The presented approach achieves good efficiency without loosing the desired correctness and completeness properties that other solvers for HFS provide.

1 Introduction and Motivation

Finite Domain (*FD*) [5] constraint solvers have been effectively applied to a great variety of problems in different application domains. Unfortunately, FD constraint solving typically suffers from an inherent weakness from the point of view of *(i)* expressive power and *(ii)* required a priori knowledge.

From the point of view of expressive power, FD constraint languages force the domains of variables to *bounded intervals* (subset of a discrete universe, often $\mathbb{Z}$), which may lead to severe limitations in real-world applications where domains of variables are often sparse sets, possibly of structured entities [12]. As a matter of fact, the use of intervals introduces an approximation in the representation of domains that may result in reduced effectiveness of constraint solving. Some constraint solvers address this issue and provide a means to deal with non-interval domains, e.g., *(i)* FD set terms [20] of SICStus allow modeling domains in terms of finite unions of intervals, and *(ii)* GNU Prolog transparently switches between sparse and interval domain representations [5].

Furthermore, forcing domains in $\mathbb{Z}$ inhibits a natural representation of structured knowledge. The recent introduction of *Finite Set* (*FS*) constraints [11] aims at overcoming such limitations because FS constraints model domains as collections of known sets. Such domains are normally expressed in terms of *set intervals* $[l..u]$ where l and u are known sets, usually of integers. A set interval $[l..u]$ represents the set of all subsets of u that contain l, i.e., $[l..u] = \{x : x \subseteq u \wedge l \subseteq x\}$. Many constraint solvers (including many CLP systems) now offer FS constraints, e.g., ECLiPSe [14], Oz [19] and B-Prolog [22].

FS constraints are explicitly derived from FD constraints and therefore they retain the appreciated efficiency of the latter, while inheriting many of their problems and limitations. In particular, available FS constraint solvers, e.g., Conjunto [11] and Cardinal [2], treat efficiently only interval domains that are defined as the convex closure of a collection of elements. For example, given a variable whose domain is $\{\{a, b\}, \{a, c\}, \{d\}\}$, the corresponding interval domain is the set interval $[\{\}..\{a, b, c, d\}]$.

Besides the mentioned expressive power limitations, it is also worth discussing another important limitation of FD-like constraint languages. The practice of using FD-like reasoning in knowledge representation has shown that domains are often unknown prior to reasoning and an important side effect of reasoning is revealing the shape of domains. In many real-world cases, domains are not available prior to computation, and they have to be acquired and/or computed; anyway, constraints over them are often known in advance. Mentioned situations are not really tractable with FD-like languages because their solvers typically attach a possibly implicit (and redundant) domain to each variable before processing constraints.

The difficulties connected with required a priori knowledge in FD-like reasoning are explicitly tackled by [10], which extends CLP($\mathcal{FD}$) to deal with incomplete knowledge on domains. The proposed system is based on the *Interactive CSP* approach [6] and it uses domains as communication channels between the CLP($\mathcal{FD}$) solver and an acquisition system.

The mentioned limitations in terms of expressive power and required a priori knowledge do not reduce the notable importance of FD-like constraint languages and solvers for their proved effectiveness in handling a great variety of problems. Nonetheless, the urge for expressive power to support modeling of complex domains that real-world applications require justifies the definition of constraint languages and solvers that trade-off efficiency with expressive power and completeness, e.g., CLP($\mathcal{SET}$) [9] and CLPS [4].

In particular, CLP($\mathcal{SET}$) provides a constraint language that subsumes most of the mentioned FD-like languages while delivering a correct and complete constraint solver, at the cost of notable inefficiency. More in details, CLP($\mathcal{SET}$), and its Java porting JSetL [18], support modeling domains in terms of generic extensional sets, usually called *Hereditarily Finite Sets* (*HFS*), that contain any kind of object (and nested finite sets in particular). Moreover, such sets can be constructed dynamically by means of common set operations and it is worth mentioning that constraint solving can take place even over partially or totally unspecified sets.

Unfortunately, the constraint solver of CLP($\mathcal{SET}$) does not exploit the information that the domain of variables provides, and it does not even explicitly express that a variable may have a domain. This leads to a very weak form of propagation and most of constraint solving is basically a *generate & test*. For example, given the following problem $x \in \{1, 2, 3, 4, 5\} \wedge x \neq 10$, the CLP($\mathcal{SET}$) solver enumerates all possible values of x before asserting that $x \neq 10$ holds.

While the expressive power and the computational features of CLP($\mathcal{SET}$) make it a good candidate to deliver a constraint language capable of addressing the mentioned issues of FD-like languages and solvers, the inefficiencies of available implementations of CLP($\mathcal{SET}$) prohibit its instant use in many problems where FD-like solvers proved their relevance. A step towards a more effective use of the CLP($\mathcal{SET}$) language is represented by CLP($\mathcal{SET}, \mathcal{FD}$) [7] which integrates FD constraints into CLP($\mathcal{SET}$) thus allowing efficient processing of the former through the use of an embedded FD solver. Advantages, however, are limited to FD constraints only: CLP($\mathcal{SET}, \mathcal{FD}$) still retains the overall constraint solving technique of CLP($\mathcal{SET}$), with no possibility to exploit domain information except for the handling of FD constraints.

The overall goal of this paper is to combine the flexibility and expressive power of CLP($\mathcal{SET}$)-like languages with the efficiency of FD-like languages, in a more general manner. In order to achieve this goal, we propose to integrate FD, FS and CLP($\mathcal{SET}$) constraints into a single, uniform constraint language and to extend the domain-driven constraint solving techniques of FD and FS to the whole solver. For this reason, we develop a new (set) constraint solver that replaces the standard CLP($\mathcal{SET}$) solver and that exploits the information on domains of variables to obtain improved efficiency.

The resulting language and solver can be viewed as a generalization of the languages of FD and FS constraints, in which domains are allowed to be general sets, containing elements of any kinds, possibly nested and partially specified.

On the other hand, the work described in this paper can be viewed as a generalization of [7], that allows retaining the expressive power of CLP($\mathcal{SET}$) while permitting the efficient processing of a larger class of CLP($\mathcal{SET}$) constraints.

The paper is organized as follows. Next section presents the language of HFS-constraints, focusing on the notion of variable domain. In Section 3, we first present the architecture of the proposed constraint solver, and then we detail the solver in terms of a set of rewrite rules which exploit FD-like reasoning within the HFS language. Finally, in Section 4 we summarize the current state of the implementation of our proposal and we point out some future work.

2 The HFS Constraint Language

This section details the syntax and semantics of the constraint language that we consider in this work. It also highlights the peculiar features that the language provides for treating domains of variables, which we exploited in the constraint solver reported in the next section.

The language described here is a minor extension of the constraint language provided by the CLP language CLP($\mathcal{SET}$), and the Java library JSetL, which delivers (most of) CLP($\mathcal{SET}$) facilities for set solving in an object-oriented language framework.

2.1 Syntax and semantics

As usual, the syntax of the language is defined by its *signature* Σ that is a triple $\langle \mathcal{V}, \mathcal{F}, \Pi \rangle$ where $\mathcal{V}$ is a fixed finite set of variables, $\mathcal{F}$ is the set of constant and function symbols, and Π is the set of *constraint* predicate symbols allowed in the language.

The set $\mathcal{F}$ of constant and function symbols is

$$\{\emptyset, \mathsf{ins}, \mathsf{int}\} \cup Z \cup F_Z \cup F_U$$

where: $\emptyset$, ins, and int are the *set constructors*; Z is the denumerable set of constants representing the integer numbers, i.e., $Z = \{0, -1, 1, -2, 2, \ldots\}$; F_Z is a set of function symbols representing operations over integer numbers, such as $+, -, *, \mathsf{div}, \mathsf{mod}$; F_U is a (possibly empty) set of user-defined constant and function symbols.

The set Π of constraint predicate symbols is

$$\{=\} \cup \Pi_S \cup R \cup \{\mathsf{set}, \mathsf{integer}\} \cup Neg$$

where: Π_S is a set of predicate symbols representing the usual set-theoretic operations, such as $\in, \subseteq, \mathsf{union}, \mathsf{inters}, ||, \mathsf{diff}, \mathsf{size}$; R is a set of predicate symbols representing the usual comparison relations over integer numbers, such as $\leq, >$; *Neg* is a set of predicate symbols representing the negated counterparts of most of other predicate symbols, such as $\neq$, $\notin$, not_integer.

A *primitive HFS-constraint* is any atomic predicate built using the symbols from the signature. A non-primitive HFS-constraint is a conjunction of primitive HFS-constraints.

The intuitive semantics of the various symbols is as follows[1]. The symbol $\emptyset$ represents the empty set. $\mathsf{ins}(t, s)$ represents the set composed of the element t union the elements of the set s, i.e., $\{t\} \cup s$. For example, $\mathsf{ins}(1, s)$, where s is an uninitialized variable, represents the (unbounded) set $\{1\} \cup s$. $\mathsf{int}(a, b, s)$ represents the set composed of the elements in the interval $[a, b]$ union the elements of the set s. If a, b are integer constants, $[a, b]$ is the set of integers $\{x : x \geq a \wedge x \leq b\}$; if a, b are ground terms denoting sets, $[a, b]$ is the set of sets $\{x : a \subseteq x \wedge x \subseteq b\}$, that is the bounded lattice induced by the subset relation $\subseteq$ having a as its greatest lower bound and b as its least upper bound.

It is worth noting that ins and int are different constructors for terms denoting the same notion of set. Thus, terms constructed using ins and int can be compared, e.g. for equality, and they can be combined. For example, $\mathsf{ins}(1, \mathsf{int}(10, 20, \mathsf{ins}(100, \emptyset)))$ represents the set containing all integers between 10 and 20 along with the integers 1 and 100. We call a term built using the set constructors ins and int an *extended set term*.

The predicates $=$ and $\in$ represent the equality and the membership relationships, respectively; the predicate union represents the union relation: $\mathsf{union}(r, s, t)$

[1] A formal discussion of these concepts is out of the scope of this paper and can be found in [7]

holds iff $t = r \cup s$; the predicate $||$ represents the disjoint relationship between two sets: $s \,||\, t$ holds iff $s \cap t = \emptyset$; and so on.

Symbols in Z are mapped to the elements of $\mathbb{Z}$, while functions in F_Z and the predicate symbols $\leq, \geq$, etc. are mapped to functions and relations over $\mathbb{Z}$ in the natural way.

We say that a HFS-constraint containing only predicates taken from $\{=, \neq, \in, \notin, \mathsf{union}, ||, \leq, \mathsf{size}, \mathsf{set}, \mathsf{integer}\}$ is in *canonical form*. As a matter of fact, [9] shows that all other predicates in Π can be defined as non-primitive constraints using the above together with the HFS theory.

Sets are defined by means of the set constructors ins and int as shown above. For the sake of simplicity, however, we will often use the following more convenient notations:

$$\{t_1 \,|\, t\}$$

as a shorthand for

$$\mathsf{ins}(t_1, t)$$

and

$$\{t_1 .. \; t_2 \,|\, t\}$$

as a shorthand for

$$\mathsf{int}(t_1, t_2, t)$$

writing $\{t_1\}$ and $\{t_1 .. \; t_2\}$, respectively, when t is the empty set. This notation is easily extended to the case of n elements (possibly mixing the ins and int constructors, e.g., $\{1, 10..20, 100\}$). With a little abuse of terminology, we call a syntactic object of the form $t_1..t_2$ a *range term*.

Various concrete language facilities are provided to make set definition simpler. For example, in JSetL, we can provide an array of elements when creating a set object; furthermore, an interval of integers can be specified at set creation by giving the lower and upper bounds of the interval. In CLP($\mathcal{SET}$), a special syntax is provided to denote a set obtained by n applications of the set constructor ins as a list of n elements with curly brackets, as in the usual mathematical notation.

Elements of a set can be values of any type (not necessarily homogeneous), including logical variables and other sets. Hence, sets can be *nested* at any level, e.g., $\{\, 1, \{\emptyset, \{a\}\}, \{\{\{b\}\}\}\,\}$. Moreover, sets can be *partially specified*, i.e., they can contain uninitialized logical variables in place of single elements.

It is worth noting that we assume that the constraints of the language deal naturally with set terms made of single elements and intervals, possibly mixed together. For example:

- $13 \in \{1..10, 15, 20..100\}$
- $\mathsf{union}(\{1, 5, 7\}, \{3..6\}, R)$
- $\{1, 3, x\} = \{1..3\}$

are all admissible HFS-constraints, whose solved forms are: false, $R = \{1, 3..7\}$, and $x = 2$, respectively. The availability of extended set terms is particularly useful to represent non-interval sets, e.g., the ones obtained from the constraint $\mathsf{diff}(\{1..10\}, \{5\}, R)$, whose solved form is $R = \{1..4, 6..10\}$.

Actually, extended set term management represents a straightforward extension of the set representation and manipulation facilities provided by CLP($\mathcal{SET}$), JSetL, and other available FD/FS constraint solvers.

2.2 Domains

Logical variables can range over the domain of HFS (*set variables*), as well as over the domain of integers (*integer variables*), and, for equality and membership constraints only, over the domain of all possible values.

Domains can be specified as HFSs and associated to variables through membership constraints, e.g., $x \in d$, where d, the domain of x, is an extended set term. This turns domains into first class abstractions of the language and naturally equips them with common set operations, thus generalizing domains from bounded intervals to very general, unbounded and potentially sparse sets.

More in detail, domains can be specified either as sets or as intervals. In the first case, the domain can be given either by enumerating all its values or it can be constructed as the result of some set operation. Moreover, the domain can be partially specified. For example, $x \in \{1, 3, 5, 7, 9\}$ states that the domain of the integer variable x is the set of the odd natural numbers less than 10, whereas $x \in \{1 \mid s\}$, with s a set variable, states that the domain of x is an unbounded set containing the value 1 plus something else not yet specified. Similarly, $r \in \{\{1\}, \{2\}, \{3\}, \{1, 2, 3\}\}$ precisely defines the domain of the set variable r as a set of four different sets.

When the domain is specified as an interval, the interval itself can be given either by specifying its lower and upper bounds, $\mathsf{int}(a, b)$, or it can be constructed as the result of some set operation, where a and b can be either integer constants (for integer variables), or known sets (for set variables). For example, the constraint $r \in \{\{1\}..\{1, 2, 3\}\}$ states that the values of r can be: $\{1\}$, $\{1, 2\}$, $\{1, 3\}$, $\{1, 2, 3\}$.

A domain d is said a *closed domain* if any element t in d is completely known (i.e., it is denoted by a ground term) or all variables possibly occurring in t have a closed domain attached; otherwise, d is an *open domain*. For instance, $\{1, x\}$, where x is an integer variable with domain $\{1..100\}$, is a closed domain. Conversely, $\{1, x\}$, with $x > 10$, and $\{1 \mid s\}$, with s a set variable with no attached domain, are (resp., bounded and unbounded) open domains. A closed domain whose elements are integer numbers is said an *integer domain*, while a closed domain whose elements are known sets is said a *set domain*.

3 Constraint Solving

3.1 The Overall Solver Architecture

In order to enhance effectiveness and efficiency of the constraint solving process, we decided to structure the proposed constraint solver into a *layered architecture*. Each layer achieves a different level of consistency at a different cost. Lower levels

are considered more effective in proving inconsistency and reducing the search space than upper levels.

More precisely the solver is structured as follows:

- Level 1 – Single constraints are processed to generate *canonical constraints* by means of deterministic rewrite rules only. Rules are triggered by the type of the primitive constraint to be handled. A very weak form of propagation of constraints is performed at this level: variable substitution only.
- Level 2 – Deterministic rules involving pairs of constraints are applied. These are all propagation rules that exploit the knowledge about variable domains, whenever possible, to reduce the size of domains or to reveal an inconsistency, according to the FD and FS approaches.
- Level 3 – Nondeterministic and labeling rules are applied. This level is highly nondeterministic and it includes the labeling of variables whose domains are known. Such level 3 rules are mostly taken from the available implementations of CLP($\mathcal{SET}$) solvers.

The overall solving process repeatedly exploits all applicable rules at a given level until a fixpoint is reached, before passing to an higher level. If no new constraints are added to the constraint store, the process steps forward to an higher lever, otherwise it restarts from level 1.

The results of levels 1 and 2 are *simplified forms* of the original constraint that we cannot prove satisfiable, much like the output of FD-like constraint solvers. Conversely, the output of level 3 enjoys the same desirable characteristics of constraint solving in CLP($\mathcal{SET}$), i.e., correctness and completeness, provided that all variables in the CSP have a closed domain attached. In fact, the result of level 3 is a finite collection $\{C_1, \ldots, C_k\}$ of constraints in *solved form* [9] which is guaranteed to be satisfiable. Moreover, the disjunction of all the constraints in solved form generated by the solver is equisatisfiable to the original constraint.

The achieved improvement of effectiveness mainly derives from the fact that the proposed layered architecture allows the user deciding which level of consistency he/she wants to achieve. As a matter of fact, the user is free to choose to *(i)* stop at any of the first two levels, with no satisfiability warrants, or *(ii)* select which (if any) variable to label if the corresponding domain is known, or optionally *(iii)* skip level 2 in order to avoid constraint propagation, thus letting more time to be spent in searching the solution space rather than in enforcing consistency. This last choice is what current implementations of CLP($\mathcal{SET}$) provides, i.e., they skip over most of the rules that we now put in level 2.

The foreseen improvement of efficiency mainly relies on the decision of postponing all (costly) nondeterministic rules at the end of the constraint solving cycle. Generally speaking, the split of rules into deterministic and nondeterministic promotes efficiency over, e.g., CLP($\mathcal{SET}$) and JSetL, because the solver is given more chance to detect inconsistency before branching any nondeterministic choice.

Finally, it is worth noting that the rewrite rules used by FD-like constraint solvers to process constraints can be entirely expressed in terms of set constraints

over (the sets that represent) the domains of variables. For example, domain reduction associated to a constraint like $x \in d \wedge x \in d'$ can be expressed by the CLP($\mathcal{SET}$) constraint $x \in D \ \wedge \ \mathsf{inters}(d, d', D)$. Thus, we can concentrate on providing efficient implementations of set constraints especially in cases where sets are completely specified, e.g., where sets represent FD or FS domains. The same set constraints, however, apply naturally also to all other cases (e.g., non-interval domains), though they are treated, in general, less efficiently.

3.2 Constraint Solving Rules

The rest of this section presents a taxonomy of the rewrite rules we employed for constraint solving. For each level of the solver, we detail the classes of rewrite rules that we used at that level, and we exemplify some rule for each class.

Many of the rewrite rules of our solver are directly derived from CLP($\mathcal{SET}$), and they have been *(i)* adapted to cope with extended set terms, and *(ii)* specialized to exploit interesting properties of special cases (e.g. ground sets). Moreover, some new rules have been introduced to deal with integer and set domains with FD- and FS-like approaches. Finally, some other rules have been introduced to handle cardinality constraints, much like in Cardinal [2].

Most of the rules are direct application of the classic set theory adapted to support HFS [9].

The rules are presented as deterministic rewrite rules that operate when respective pre-conditions are met:

$$\frac{\textit{pre-conditions}}{\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}}$$

where $C_1, \ldots, C_n$ and $C'_1, \ldots, C'_m$ (n, $m \geq 0$) are primitive HFS-constraints in the constraint store, and $\{C_1, \ldots, C_n\} \rightarrow \{C'_1, \ldots, C'_m\}$ represents the changes in the constraint store caused by the application of the rule.

In the rest of this section, we adopt the following notation:

- t, t_i: any term (either ground or not);
- c, c_i: any ground term (either integer or not);
- i, i_i: integer constants;
- v, v_i: any non-ground term;
- X, Y, Z, N, M, ... (uninitialized) variables; and
- s, r, s_i: any set (either ground or not).

We also introduce the function $\mathsf{dom}(X)$, which returns the closed domain currently associated with X, or $-\infty..\ +\infty$ if the domain of X is open or X has no domain attached. The details on how $\mathsf{dom}(.)$ is concretely implemented is out of the scope of this paper, but it is worth noting that the domain of a variable can be equally stored as an attribute of the variable itself, or as an appropriate constraint that links a variable to its domain.

$\mathsf{dom}(X)$ is defined for any X and we need to complement it with two other predicates to perform the common task of deciding whether X is an integer variable or a set variable. In particular, $\mathsf{int\_dom}(d)$ holds if d is an *integer domain*, and $\mathsf{set\_dom}(d)$ holds if d is a *set domain*.

Level 1

This level performs a simplification of constraints by means of six classes of rewrite rules.

Ground cases – Solve constraints whose arguments are all ground terms. For example:

- *set membership test*

$$\frac{c \neq c_1 \text{ and } \ldots \text{ and } c \neq c_n, c_i \text{ not range terms}}{\{c \in \{c_1, \ldots, c_n \mid X\}\} \rightarrow \{c \in X\}} \tag{1}$$

- *interval membership test*

$$\frac{i \geq i_1, i \leq i_2}{\{i \in \{i_1..i_2\}\} \rightarrow \{\}} \tag{2}$$

Similar rules apply to other constraints that deal with both integer and set intervals, such as union, intersection and difference. These operations are fundamentals for the implementation of the FD and FS constraint solving techniques as illustrated for instance in [11] and [2]. For this reason it is crucial that these rules are implemented as efficiently as possible. This is the case, e.g., of $\mathsf{inters}(1\,..\,10, 5\,..\,20, D)$, which is easily solved because both intervals are completely known. Conversely, a constraint like $\mathsf{inters}(\{1, X\}, \{2, Y\}, D)$ is managed with the more general, though less efficient, rules of level 3.

Special cases – Identify simple cases, in which not all terms are necessarily ground, e.g, one term is ground and/or one term is a set whose elements are all variables and/or terms are singleton sets. For example:

- *empty set*

$$\frac{}{\{t \in \emptyset\} \rightarrow \mathsf{fail}} \tag{3}$$

- *singleton set*

$$\frac{t_2 \text{ not a range term}}{\{t_1 \in \{t_2\}\} \rightarrow \{t_1 = t_2\}} \tag{4}$$

Generation of canonical constraints – Primitive constraints that are not in a canonical form and that can not be further simplified are rewritten to semantically equivalent canonical constraints. For example:

- *subset*

$$\frac{}{\{X \subseteq Y\} \rightarrow \{\mathsf{union}(X, Y, Y)\}} \tag{5}$$

Inference of types – Provide additional constraints to ensure type coherence. For example:

- *integer variable*

$$\frac{\mathsf{int\_dom}(s)}{\{X \in s\} \rightarrow \{X \in s, \mathsf{integer}(X)\}} \tag{6}$$

Inference of cardinalities – Provide additional constraints to ensure coherence of the cardinality of sets. For example:

– *set variable*

$$\frac{\mathsf{set\_dom}(\{s_1 \,..\, s_2\}), |s_1| = a, |s_2| = b}{\{X \in \{s_1 \,..\, s_2\}\} \rightarrow \{X \in \{s_1 \,..\, s_2\}, \mathsf{size}(X, N), N \in \{a \,..\, b\}\}} \tag{7}$$

Inference of domains – Gather information on domains that is not explicitly provided. For example:

– *cardinality of a partially specified set*

$$\frac{\begin{array}{l} m = \text{ number of distinct ground terms in} \{t_1, \ldots, t_n\} \text{ or} \\ \qquad m = 1 \text{ if } t_1, \ldots, t_n \text{ are all non-ground terms} \end{array}}{\{\mathsf{size}(\{t_1, \ldots, t_n\}, N)\} \rightarrow \{N \in \{m..n\}\}} \tag{8}$$

For example, $\mathsf{size}(\{X, Y, 1\}, N)$ generates $N \in \{1..3\}$, while $\mathsf{size}(\{1, 2, 3\}, N)$ is rewritten to $N \in \{3..3\}$, i.e., $N = 3$.

All cases that are not tackled by the mentioned classes of rules are considered *irreducible* at level 1 and they are therefore shipped to level 2. In particular, constraints like $t \in \{t_1, \ldots, t_n \mid s\}$, with $n \geq 1$, s either variable or $\emptyset$, and either t or $t_1, \ldots, t_n$ non-ground terms are passed unaltered to level 2 (note that constraints of the form $X \in \{t_1, \ldots, t_n \,|\, s\}$ represent sort of domain declarations that are managed at level 2).

Remark 1. The outcome of level 1 is in canonical form, which eases the tasks of level 2 because it can address only couples of constraints in canonical form. While this feature of the output of level 1 is crucial to limit the potential explosion of rules at level 2, on the other hand it may introduce some inefficiencies in the overall constraint solving process. As a matter of fact, the relation represented by a non-canonical constraint may become no longer evident after the constraint have been replaced by the (equivalent) conjunction of primitive constraints in canonical form. Hence, some optimizations and/or constraint inferences that apply to the original constraint are hardly applicable to the transformed constraint. For example, the non-canonical constraint $\mathsf{inters}(X, Y, Z)$ is replaced by the equivalent canonical constraint $\mathsf{union}(A, Z, X), \mathsf{union}(B, Z, Y), A \,||\, B$; if after this replacement, X, Y, Z get instantiated to ground set terms, nevertheless the solver can not apply the efficient processing rules provided for dealing with intersection over ground sets. Finding the appropriate tradeoff between reducing the number of cases to be managed by the solver's upper levels and delaying as long as possible elimination of non-canonical constraints is left as an open problem for future work.

Level 2

Rules at this level are mainly intended to perform an FD-like reasoning on domains of variables. This level provides also rules to perform simple consistency checks between pairs of constraints (e.g., between type constraints).

Domains management – Add/remove elements from the domains of variables, merge sparse domain information of single variables, test for domain membership. For example:

– *domain update*

$$\frac{\mathsf{set\_dom}(d), \mathsf{update\_dom}(d, c) = d'}{\{c \in X, X \in d\} \rightarrow \{X \in d'\}} \tag{9}$$

where $\mathsf{update\_dom}(d, c)$ returns the subset d' of the set domain d whose elements contain the constant element c, i.e.,

$$d' = \{s : s \in d \wedge c \in s\}.$$

For example:

- $\mathsf{update\_dom}(\{\emptyset..\{1,2,3\}\}, 1)) = \{\{1\}..\{1,2,3\}\}$
- $\mathsf{update\_dom}(\{\emptyset..\{1,2,3\}\}, 4)) = \emptyset$
- $\mathsf{update\_dom}(\{\{1,2\},\{1,3\},\{2,3\}\}, 1) = \{\{1,2\},\{1,3\}\}$
- $\mathsf{update\_dom}(\{\{1,2\}, a\}, 1) = \{\{1,2\}\}$.

Similarly, the processing of disequalities may cause elements to be removed from a domain:

$$\frac{s, t \text{ ground}}{\{X \neq t, X \in s\} \rightarrow \{X \in D, \mathsf{diff}(s, \{t\}, D)\}} \tag{10}$$

In particular, when s is an integer interval $\{i_1..i_2\}$ and t is an integer belonging to the interval, distinct from i_1 and i_2, the resulting new domain D is $\{i_1..t-1, t+1..i_2\}$.

– *domain merge*

$$\frac{r, s \text{ ground}}{\{X \in s,\ X \in r\} \rightarrow \{X \in D, \mathsf{inters}(s, r, D)\}} \tag{11}$$

Rules (10 and (11) are applicable for any (ground) s and r, which requires diff and inters to work equally with interval and non-interval sets. As an example, $\mathsf{inters}(\{\{1\}..\{1,2,3\}\}, \{\{1,2\},\{2,3\}\}, X)$ holds for $X = \{\{1,2\}\}$.

– *domain test*

$$\frac{c \notin \mathsf{dom}(X_k) \text{ and } \ldots \text{and } c \notin \mathsf{dom}(X_n)}{\{c \in \{X_1, \ldots, X_{k-1}, X_k, \ldots, X_n\}\} \rightarrow \{c \in \{X_1, \ldots, X_{k-1}\}\}} \tag{12}$$

This rule removes all variables whose domains do not contain c. It is worth noting that this rules has an interesting special case for $k = 1$ that leads to fail via $\{c \in \{X_1, \ldots, X_n\}\} \rightarrow c \in \emptyset$.

As an example of application of rule (12), which shows some improvements of our approach on CLP($\mathcal{SET}$) (and CLP($\mathcal{SET}, \mathcal{FD}$)), let us consider the following HFS-constraint:

$$R = \{X1, X2, X3, X4, X5\}, 15 \in R, R \subseteq \{1, 3, 5, 7, 9, 11\}$$

After propagating equality for R over the other two constraints through variable substitution, the solution of the $\subseteq$ constraint forces all variables $X1$, $X2$, $X3$, $X4$, $X5$ to get the set {1,3,5,7,9,11} as their domain; then the constraint $15 \in \{X1, X2, X3, X4, X5\}$ is reduced by rule (12) to $15 \in \emptyset$, hence to false. Note that CLP($\mathcal{SET}$) addresses this constraint with a *generate & test* approach–generates all possible values for R and then test them using the $\in$ constraint.

Remark 2. The condition that the sets involved in rules (10) and (11) must be ground in not strictly necessary. Allowing these rules to be applied even to more general cases, however, may cause troubles to arise:

- with termination of the constraint solving algorithm;
- with efficiency of the rewriting process, since the solution of constraints like diff and inters applied to general sets may generate (through non-determinism) a large number of (possibly redundant) solutions.

For these reasons we prefer for now to restrict applicability of these rules to the ground case only. A more precise analysis of cases in which it may be convenient to allow non-ground cases to be dealt with is left for future work. As an example of one such case, consider the constraint $X \in \{1, 3, 5, Y\}$ which states that the domain of X contains the values 1, 3, and 5, and possibly another value Y which is left unspecified for the moment. Assume that the constraint store contains also the constraint $Xneq3$. If we remove the groundness condition of rule (10), then we can apply this rule and we get the new constraints $X \in \{1, 5, Y\}, Y \neq 3$. Thus, application of rule (10) allows us to get a reduced domain for X although it is not completely specified.

Type clash – Identify clashes in type constraints, e.g., $\mathsf{set}(X) \wedge \mathsf{integer}(X)$ immediately fails.

Size management – Make sure the cardinality of a set is always unique:

- *size uniqueness*

$$\frac{}{\{\mathsf{size}(s, N), \mathsf{size}(s, M)\} \rightarrow \{\mathsf{size}(s, N), N = M\}} \tag{13}$$

Besides the mentioned cases $X \in s \wedge X \in r$ and $X \in s, X \neq t$, level 2 deals with all combinations of canonical constraints not eliminated at level 1 that share some variable. In particular, level 2 treats the combinations of $X \in s$ with:

- $X \notin r$;
- $X \leq e$, $size(s, X)$ (X integer variable);
- $\mathsf{union}(X, Y, Z)$, $\mathsf{union}(Y, Z, X)$, $X \,||\, Y$ (X set variable).

Moreover, it is worth noting that the case $X \in s \wedge X \leq e$, with e arithmetic expression, fires the application of common FD-like rules that cause an FD-like propagation.

Finally, the rules of level 2 can exploit an interesting optimization of implementation. Most of reasonable implementations of the proposed constraint solving approach would likely store all knowledge about the domain of X in an attribute of X itself. Therefore, rules of level 2 may exploit this to avoid scanning of the entire constraint store while looking for constraints of the form $X \in d$ in order to associate X with its domain.

Level 3

This level groups all rewrite rules that involve nondeterminism. In particular rules dealing with membership constraints of the form $X \in s$ allow forcing assignment to the specified variables of values from their domains, leading to a chronological backtracking search of the space of solutions.

Nondeterministic choices – Open nondeterministic branches. For example:

- *(set inequality)*

$$\frac{}{\{s \neq r\} \rightarrow \{Z \notin s, Z \in r\} \text{ or } \{s \neq r\} \rightarrow \{Z \in s, Z \notin r\}} \tag{14}$$

  where s and r are (any) sets.

Enumeration – Label variables with appropriate values. For example:

- *(set membership)*

$$\frac{t_1 \text{ not a range term}}{\{X \in \{t_1 \mid s\}\} \rightarrow \{X = t_1\} \text{ or } \{X \in \{t_1 \mid s\}\} \rightarrow \{X \in s\}} \tag{15}$$

Note that other forms of solution enumeration are obtained from the treatment of other constraints such as $\mathsf{union}(X, Y, \{1, 2, 3\})$ which assigns to X and Y all possible subsets whose union yields $\{1, 2, 3\}$.

From a pragmatic point of view, the treatment of this kind of constraints, that leads to a generalized notion of labeling, could be explicitly activated or deactivated by the user on request.

4 Discussion

The presented work has been mainly performed on a theoretical basis and just some preliminary experiments are available. During the preliminary inclusion of HFS in a practical solver, we faced some technical difficulties.

The architecture of the solver that we sketched in Section 3 clearly shows that the solver embeds FD and FS constraint solving as particular cases. Actually, one of the main duties of level 2 is to identify FD and FS sub-problems and treat them accordingly. At the implementation level, this can be obtained either by a single solver with specialized subparts or via solver cooperation [13], e.g. using a master/slave architecture as discussed in [3], where a master solver invokes one or more existing slave solvers when needed.

The master/slave approach has been adopted in the CLP($\mathcal{SET}, \mathcal{FD}$) proposal [7] where FD constraint solving is made available within CLP($\mathcal{SET}$) by exploiting the facilities of an existing FD solver (specifically the FD solver of SICStus in the current implementation). Conversely, the integration of FD constraints within JSetL, our Java implementation of CLP($\mathcal{SET}$), is based on a single solver in which rewrite rules dealing with FD constraints have been developed from scratch and embedded in the general solver that deals with all other (set) constraints.

The constraint technique described in this paper has not been fully implemented yet. However, previous experiences with the integration of FD and CLP($\mathcal{SET}$), both within the Prolog interpreter {log} [17] and within the Java library JSetL, gave us some feedback about the feasibility and the possible performance improvements of this approach. In particular, the current version of JSetL, named JSetL($\mathcal{FD}$) [1, 16], provides most of the CLP($\mathcal{SET}$) facilities together with basic FD constraint solving facilities. It also supports the constraint `size` and it provides an implementation of the constraint `all_different` based on the well known approach of Hall sets [15].

The preliminary experiments that we performed using this implementation show that the use of FD techniques within our solver strongly enhances its performances on problems that can modeled as simple FD problems. This confirms the encouraging results obtained with CLP($\mathcal{SET}, \mathcal{FD}$) and documented in [8].

Table 1 shows the results we obtained using JSetL($\mathcal{FD}$) on the standard n-queens problem. On this problem JSetL($\mathcal{FD}$) exhibits performances that are comparable to (or even better than) those of existing FD solvers, in particular the one provided by SWI Prolog [21].

n	Count of solutions	JSetL($\mathcal{FD}$) Binary decomposition	JSetL($\mathcal{FD}$) `all_different`	SWI Prolog
7	40	250ms	187ms	510ms
8	92	1,109ms	442ms	2,750ms
9	352	5,703ms	1,640ms	13,810ms
10	724	30,375ms	5,687ms	75,520ms
11	2,680	243,047ms	20,750ms	43,150ms
12	14,200	$>$ 3 min	105,391ms	$>$ 3 min

Table 1. Enumeration of all solutions of the n-queens problem

For the near future, we plan to extend the current implementation JSetL($\mathcal{FD}$) to completely include the constraint solving technique described in this paper. This will be achieved by:

- Extending the current representation of sets by allowing JSetL($\mathcal{FD}$) to support extended set terms that mix single set elements and intervals;
- Providing efficient implementation of all constraints in the ground cases, including those intended to handle integer and set intervals;
- Implementing rules for FS constraints to effectively manage set domains, as provided, e.g., by Conjunto and Cardinal; and
- Structuring the whole solver in terms of the layered architecture described in Section 3.

References

1. Amadini, R. (2007) *Definizione e trattamento del vincolo di cardinalità insiemistica nella libreria JSetL* Tesi di Laurea, Dipartimento di Matematica, Università di Parma.
2. Azevedo, F. (2007) Cardinal: A Finite Sets Constraint Solver, *Constraints*, 12(37):93–129.
3. Bergenti, F., Panegai, E., and Rossi, G. (2006) A Master-Slave Architecture to Integrate Sets and Finite Domains in Java, Presented at CILC'06 – Convegno Italiano di Logica Computazionale, Bari.
4. Bouquet, F., Legeard, B., and Peureux, F. (2002) CLPS-B - a constraint solver for B. In *Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 2280 of LNCS, Springer Verlag, pp. 188–204.
5. Codognet, P., and Diaz, D. (1996) Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226.
6. Cucchiara, R., Gavanelli, M., Lamma, E., Mello, P., Milano, M., Piccardi, M. (online) Extending the CSP Model to Cope With Partial Information. Available at: `http://lia.deis.unibo.it/Research/Areas/icsp`
7. Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2003) Integrating Finite Domain Constraints and CLP with Sets. In D. Miller, ed., *Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229.
8. Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2006) Constraint Logic Programming Language for Effective Programming with Sets and Finite Domains. Research Report "Quaderno del Dipartimento di Matematica", 437, Università di Parma.
9. Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. (2000) Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931.
10. Gavanelli M., Lamma E., Mello P., and Milano, M. (2005) Dealing with incomplete knowledge on CLP(FD) variable domains, *ACM Transactions on Programming Languages and Systems*, 27(2):236–263.
11. Gervet, C. (1997) Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244.
12. Gervet, C. (2006) Constraints over Structured Domains In Rossi, F. van Beek, P., and Walsh, T. (Eds.), Handbook of Constraint Programming, Elsevier.
13. Hofstedt, P. (2000) Cooperating Constraint Solvers. In Dechter, R. (Ed.), *International Conference on Principle and Practice of Constraint Programming*, Vol. 1894 of LNCS, Springer Verlag, 520–524.
14. IC Parc (2003) *The ECLiPSe Constraint Logic Programming System.* London. `www.icparc.ic.ac.uk/eclipse/`.
15. Leconte, M. (1996) A bounds-based reduction scheme for constraints of difference. In Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning, pp. 19–28
16. Pandini, D. (2008) *Progettazione e realizzazione in Java di un risolutore di vincoli su domini finiti* Tesi di Laurea, Dipartimento di Matematica, Università degli Studi di Parma.
17. Rossi, G. (2005) *The {log} Constraint Logic Programming Language.* `prmat.math.unipr.it/~gianfr/setlog.Home.html`.

18. Rossi, G., Panegai, E., and Poleo, E. (2007) JSetL: A Java Library for Supporting Declarative Programming in Java *Software-Practice & Experience*, 37:115-149.
19. van Roy, P. (Ed.) (2005) Multiparadigm Programming in Mozart/Oz Lecture Notes in Computer Science 3389 Springer 2005, ISBN 3-540-25079-4.
20. Swedish Institute of Computer Science. *The SICStus Prolog Home Page*. `www.sics.se`.
21. Wielemaker, J. (2004) *SWI-Prolog Reference Manual (Version 5.4)*. University of Amsterdam.
22. Zhou, N-F. (2005) *B-Prolog User's Manual (Version 6.8)*. Afany Software.