Integrating Finite Domain Constraints and CLP with Sets

A. Dal Palù[1], A. Dovier[1], E. Pontelli[2], and G. Rossi[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica. (dalpalu|dovier)@dimi.uniud.it
[2] New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu
[3] Univ. di Parma, Dip. di Matematica. gianfr@prmat.math.unipr.it

Abstract. We propose a semantically well-founded combination of the constraint solvers used in the constraint programming languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). This work demonstrates that it is possible to provide efficient executions (through CLP($\mathcal{FD}$) solvers) while maintaining the expressive power and flexibility of the CLP($\mathcal{SET}$) language. We develop a combined constraint solver and we show how static analysis can help in organizing the distribution of constraints to the two constraint solvers.

1 Introduction

Several proposals aimed at developing declarative programming frameworks that incorporate different types of *set-based primitives* have been presented (e.g., [1, 10, 13, 4, 5, 2]). These frameworks provide a high level of data abstraction, where complex algorithms can be encoded in a natural fashion, by directly using the popular language of set theory. These features make this type of languages particularly effective for modeling and rapid prototyping of algorithms. One of the main criticisms moved to these approaches is that the intrinsic computational costs of some set operations (e.g., the NP-completeness of set-unification) make them suited for fast prototyping but not for the development of efficient solutions. On the other hand, various proposals have been made to introduce constraint-based language primitives for dealing with simple forms of sets—i.e., *finite* subsets of a predefined *domain* (typically $\mathbb{Z}$). Constraint-based manipulation of finite domains provides the foundations of the popular CLP($\mathcal{FD}$) framework [15, 3], that has been demonstrated well-suited for the encoding and the fast resolution of combinatorial and optimization problems [11].

The overall goal of this work is to develop a bridge between the expressive power and the high level of abstraction offered by constraint solving over generic hereditarily finite sets—as possible in CLP($\mathcal{SET}$) [5]—and the efficient use of constraint propagation techniques provided by constraint solving over finite domains—as possible in CLP($\mathcal{FD}$). In order to accomplish these results, we extend the CLP($\mathcal{SET}$) framework by introducing integer constants and integer intervals, as a natural extension of the existing constraint handling capabilities—the new language is called CLP($\mathcal{SET}$)$^{\mathsf{int}}$. We develop novel constraint solving algorithms that allow the CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$) solvers to communicate and cooperate in verifying the satisfiability of constraint formulae over the combined structure (finite domains + hereditarily finite sets). We also present static analysis strategies for CLP($\mathcal{SET}$) programs which make it possible to automatically discover situations in which the CLP($\mathcal{FD}$) constraint solver can be used in place of the CLP($\mathcal{SET}$) solver to check constraint satisfiability. Moreover, the analyzer can detect entire fragments of CLP($\mathcal{SET}$) programs that can be translated to equivalent CLP($\mathcal{FD}$) programs and as such executed more efficiently.

This work represents a natural evolution of the previous research on programming with sets, starting from the logic language {log} [4], later developed as a constraint logic programming language (CLP($\mathcal{SET}$) [5]) and endowed with more set-theoretic primitives. The methodology proposed in this paper extends the ideas in CLP($\mathcal{SET}$) in two directions. On one hand, we provide CLP($\mathcal{SET}$) with a methodology to execute programs more efficiently, by relying on the fast propagation mechanisms developed for CLP($\mathcal{FD}$); on the other hand, we allow CLP($\mathcal{SET}$) to become a high-level language to express constraint-based manipulation of CLP($\mathcal{FD}$) domains. A similar idea is behind the work by Flener et al. [6], where the authors compile the language ESRA, that includes set-based primitives, into the constraint-based language OPL. Gervet in [7] introduces set operations over finite domains in the CLP language ECLiPSe [8]. In [17] the authors propose a tuple-based treatment of sets in ECLiPSe and compare it with Gervet's approach. These two approaches, however, do not allow a complete and easy handling of programs involving sets.

2 CLP($\mathcal{SET}$) with Intervals

The signature Σ upon which CLP($\mathcal{SET}$) [5] is based is composed of the set $\mathcal{F}$ of function symbols, that includes $\emptyset$ and $\{\cdot \mid \cdot\}$, the set $\Pi_C = \{=, \in, \cup_3, ||, \mathsf{set}\}$ of constraint predicate symbols, and a denumerable set $\mathcal{V}$ of variables. The intuitive semantics of the various symbols is the following: $\emptyset$ represents the empty set, while $\{t \mid s\}$ represents the set composed of the elements of the set s plus the element t (i.e., $\{t \mid s\} = \{t\} \cup s$). $=$ and $\in$ represent the equality and the membership relations. $\cup_3$ represents the union relation: $\cup_3(r, s, t)$ holds if $t = r \cup s$. The predicate $||$ captures the disjoint relationship between two sets: $s||t$ holds if and only if $s \cap t = \emptyset$. The constraint predicate set is used to enforce the multi-sorted nature of the language; terms are separated into two sorts, one representing set terms (called `Set`) and one representing non-set terms. The predicate set checks whether a term belongs to the sort `Set` (e.g., $\mathsf{set}(\{a, b\})$ holds while $\mathsf{set}(1)$ does not). The semantics of CLP($\mathcal{SET}$) has been described with respect to a predefined structure $\mathcal{A}_{\mathcal{SET}}$ [5].

We will use the notation $\{t_1, t_2, \ldots, t_n \mid t\}$ for $\{t_1 \mid \{t_2 \mid \cdots \{t_n \mid t\} \cdots\}\}$ and the notation $\{t_1, t_2, \ldots, t_n\}$ as a shorthand for $\{t_1 \mid \{t_2 \mid \cdots \{t_n \mid \emptyset\} \cdots\}\}$. The *primitive constraints* in CLP($\mathcal{SET}$) are all the positive literals built from symbols in $\Pi_c \cup \mathcal{F} \cup \mathcal{V}$. A *($\mathcal{SET}$-)constraint* is a conjunction of primitive constraints and negation of primitive constraints of CLP($\mathcal{SET}$). Other basic set-theoretic operations (e.g., $\cap$, $\subseteq$) are easily defined as $\mathcal{SET}$-constraints. CLP($\mathcal{SET}$) provides also a syntactic extension called *Restricted Universal Quantifiers (RUQs)*, that allows one to write subgoals of the form $(\forall X_1 \in S_1) \cdots (\forall X_n \in S_n)(C \wedge B)$ where C is a $\mathcal{SET}$-constraint and B is a CLP($\mathcal{SET}$) program goal. The semantics of $(\forall X \in S)G$ is simply $\forall X\, (X \in S \rightarrow G)$. The implementation of RUQs is accomplished via program transformation—generating two recursive CLP($\mathcal{SET}$) clauses using set terms and constraints [4].

In this paper we assume that the constant symbols $0, 1, -1, 2, -2, \ldots$ are in $\mathcal{F}$. Let us call these terms *integer constants* (or, simply, *integers*). Let us also assume that a total order $\prec$ is defined on $T(\mathcal{F})$ (see [5] for details) such that $\ldots - 2 \prec -1 \prec 0 \prec 1 \prec 2 \prec \ldots$. We say that $s \preceq t$ iff $s \prec t$ or $s \equiv t$. When clear from the context, we denote with $x + y$ $(x - y)$ the integer constant corresponding to the sum (difference) of the values of the two integer constants x and y. We will also use $<$ and $\leq$ with the usual meaning. These terms are seen as different, uninterpreted, constant terms. Since they are not interpreted, the constraint domain for CLP($\mathcal{SET}$) does not need to be changed.

We also introduce in $\mathcal{F}$ the binary function symbol int. A term $\mathsf{int}(t_i, t_f)$, where t_i and t_f are integer constants, is called an *interval term.* $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\{t_i, t_i + 1, \ldots, t_f - 1, t_f\}$. If $t_f \prec t_i$, then $\mathsf{int}(t_i, t_f)$ is a shorthand for the set term $\emptyset$. We only allow variables or integer constants as arguments of the function symbol int. To start, however, we require the terms t_i, t_f in $\mathsf{int}(t_i, t_f)$ to be non-variable. This restriction will be relaxed in the successive sections. Terms of the form $\mathsf{int}(t_1, t_2)$ with either t_1 or t_2 not integer constants or variables are considered ill-formed terms. Occurrences of ill-formed terms in a constraint cause the CLP($\mathcal{SET}$) solver to return a false result. Moreover, if it is not ill-formed, $\mathsf{set}(\mathsf{int}(t_1, t_2))$ holds. CLP($\mathcal{SET}$) with these extensions is referred to as *CLP($\mathcal{SET}$) with intervals* or, simply, CLP($\mathcal{SET}$)$^{\mathsf{int}}$.

3 CLP($\mathcal{SET}$)$^{\mathsf{int}}$: Constraint Solving modifying $SAT_{\mathcal{SET}}$

CLP($\mathcal{SET}$) provides a sound and complete constraint solver ($SAT_{\mathcal{SET}}$) to verify satisfiability of $\mathcal{SET}$-constraints. Given a constraint C, $SAT_{\mathcal{SET}}(C)$ non-deterministically transforms C either to false (if C is unsatisfiable) or to a finite collection $\{C_1, \ldots, C_k\}$ of constraints in *solved form* [5]. A constraint in solved form is guaranteed to be satisfiable in the considered structure $\mathcal{A}_{\mathcal{SET}}$. Moreover, the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is *equi-satisfiable* to C. For instance, $SAT_{\mathcal{SET}}(\{1, 2 \,|\, X\} = \{1 \,|\, Y\} \wedge 2 \notin X)$ returns the three solved form constraints:

- $Y = \{2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$
- $X = \{1 \,|\, N\} \wedge Y = \{2 \,|\, N\} \wedge \mathsf{set}(N) \wedge 2 \notin N$
- $Y = \{1, 2 \,|\, X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$

where N is a new variable—to be treated as an existentially quantified variable. A detailed description of the CLP($\mathcal{SET}$) solver can be found in [5]; a sound and complete implementation of $SAT_{\mathcal{SET}}$ is included in the $\{log\}$-interpreter [12].

Interval terms can be treated simply as syntactic extensions and automatically removed before processing the constraints. The simplest solution is to replace each interval term $\mathsf{int}(t_i, t_f)$ with the corresponding extensional set term

$\{t_i, t_i + 1, \ldots, t_f\}$. Obviously, this is a rather inefficient solution, as it may lead to an explosion in the size of the constraint.

A more effective way to handle constraints involving interval terms is to modify the rewriting procedures of the CLP($\mathcal{SET}$) constraint solver. The modifications should guarantee that atomic constraints involving intervals (e.g., $t \in \mathsf{int}(t_i, t_f)$) have the expected semantics, but without having to expand interval terms into the corresponding set terms. The modified rewriting rules for $\in$ constraints are shown in the figure below. We omit the details of the procedures for the other constraint predicates for space reasons.

(1)	$s \in \emptyset \wedge C'$	$\mapsto$ false
(2)	$r \in \{s \,\vert\, t\} \wedge C'$	$\mapsto$ $C' \wedge r = s$ *or* $C' \wedge r \in t$
(3)	$t \in X \wedge C'$	$\mapsto$ $X = \{t \,\vert\, N\} \wedge \mathsf{set}(N) \wedge C'$
(4.1)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_f \prec t_i, t_i, t_f$ *integers*	$\mapsto$ false
(4.2)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_n \preceq t_f, t_i, t_f, t_n$ *integers*	$\mapsto$ C'
(4.3)	$X \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f, t_i, t_f$ *integers* *and* X *is a variable*	$\mapsto$ $C' \wedge X = t_i$ *or* $C' \wedge X \in \mathsf{int}(t_i + 1, t_f)$
(4.4)	$t_n \in \mathsf{int}(t_i, t_f) \wedge C'$ $t_i \preceq t_f, t_i, t_f, t_n$ *integers* $t_n \prec t_i$ *or* $t_f \prec t_n$	$\mapsto$ false
(4.5)	$r \in \mathsf{int}(t_i, t_f) \wedge C'$ r *is neither a variable* *nor an integer*	$\mapsto$ false

Although this solution avoids exploding interval terms to set terms and ensures good efficiency in processing each individual membership constraint over intervals, it is still unsatisfactory whenever the constraint involves more than one interval term. For example, consider the $\mathcal{SET}$-constraint: $X \in \mathsf{int}(1, 1000) \wedge X \in \mathsf{int}(1001, 2000)$. According to the above definition of the membership rewriting procedure, solving this constraint will cause the constraint solver to repeatedly (through backtracking) generate at least 1000 integer constants to solve the first or the second constraint, before concluding that the constraint is unsatisfiable. This problem derives from the fact that the constraint solver does not fully exploit the semantic properties of interval constraints.

4 CLP($\mathcal{FD}$): Language and Constraint Solving

The language CLP($\mathcal{FD}$) [15, 3, 11] is an instance of the CLP scheme, and it is particularly suited to encode combinatorial problems. The constraint domain used by CLP($\mathcal{FD}$) contains an alphabet of constraint predicate symbols $\Pi_C = \{\{\in m..n\}, =, \neq, <, \leq, >, \geq\}$ (see, e.g. [9]) and an interpretation structure $\mathcal{A}_{\mathcal{FD}}$, whose interpretation domain is the set $\mathbb{Z}$ of the integer numbers and the interpretation function $(\cdot)^{\mathcal{FD}}$ maps symbols of the signature to elements of $\mathbb{Z}$ and to functions and relations over $\mathbb{Z}$ in the natural way. In particular, the interpretation of the constraint $u \in m..n$ is: $u \in^{\mathcal{FD}} m..n$ holds if and only if $m \leq u \leq n$.[4]

[4] $u \in m..n$ is expressed as `u in m..n` in SICStus and as `u::m..n` in ECLiPSe.

The constraint predicates $\{\in m..n\}$ are used to identify intervals representing the *domains* of the variables in the problem. Additional constraint predicates are often provided in CLP($\mathcal{FD}$) languages, e.g., cumulative constraints and labeling, but for the purpose of this paper we will only focus on the primitive constraints mentioned above. A *$\mathcal{FD}$-constraint* is a conjunction of literals built from symbols in the given signature.

Let $SAT_{\mathcal{FD}}$ be a constraint solver for the $\mathcal{FD}$-constraint domain. Here we consider $SAT_{\mathcal{FD}}$ as a black-box, i.e., a procedure that takes as input a $\mathcal{FD}$-constraint and returns a set of $\mathcal{FD}$-constraints. The resulting constraints cover the same solution space as the input constraint, but they are simplified constraints. The CLP($\mathcal{FD}$) solvers adopted in SICStus [14] and ECLiPSe [8] are incomplete solving procedures; e.g., given the goal

```
| ?- X in 1..2, Y in 1..2, Z in 1..2, X #\= Y, X #\= Z, Y #\= Z.
```

the CLP($\mathcal{FD}$) solvers provided by SICStus and ECLiPSe are not able to find out the inconsistency (actually they leave the given constraint unchanged). A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space. E.g., if we conjoin the goal above with the atom `labeling([X,Y,Z])` (`labeling([], [X,Y,Z])` in SICStus) then the constraint solver will successfully determine the unsatisfiability of the constraint.

Thus, given a constraint C, using CLP($\mathcal{FD}$), we can either obtain a constraint C' implying C, possibly not in solved form and possibly unsatisfiable, or, via labeling, a solved form—possibly ground—solution. This can be compared with the case of CLP($\mathcal{SET}$), where we always obtain a disjunction of solved form satisfiable constraints C'_i (or false if C is unsatisfiable) (see Section 3).

5 CLP($\mathcal{SET}$)$^{\mathsf{int}}$: Combining Solvers

The solution we propose is to exploit the CLP($\mathcal{FD}$) constraint solver within the CLP($\mathcal{SET}$) constraint solver to solve membership constraints over intervals, by mapping the $\in$ constraints of CLP($\mathcal{SET}$) to $\{\in m..n\}$ constraints of CLP($\mathcal{FD}$), as well as equations and disequations over integers, which may occur in a $\mathcal{SET}$-constraint. Extensions that deals with $<$ and $\leq$ are discussed in Section 6. In the specific implementation described here we make use of the constraint solver offered by SICStus Prolog [14], though the proposed technique is general and applicable to other CLP($\mathcal{FD}$) solvers. Let us assume that the two languages make use of the same set of variables and the same representation of integer constants. The function τ that translates atomic $\mathcal{SET}$-constraints to $\mathcal{FD}$-constraints is defined as follows:

$$\tau(c) = \begin{cases} s \in t_i..t_f & \textit{if } c \equiv s \in \mathsf{int}(t_i, t_f) \;\; (\textit{or } \; s \in \{t_i, t_i+1, \ldots, t_f\}) \\ s = t & \textit{if } c \equiv s = t \textit{ or } c \equiv s \;\mathsf{is}\; t \\ s \neq t & \textit{if } c \equiv s \neq t \\ \mathsf{true} & \textit{otherwise} \end{cases}$$

where s, t_i, t_f, t are either variables or integer constants. In concrete syntax, the CLP($\mathcal{SET}$) constraint $s \in \mathsf{int}(m, n)$ will correspond to the SICStus CLP($\mathcal{FD}$)

constraint `s in m..n`, whereas $s \neq t$ and $s = t$ will correspond to `s #\= t` and `s #= t`, respectively. τ can be naturally extended to conjunctions of primitive constraint. We will also refer to the inverse function τ^{-1}, where $\tau^{-1}(s \in t_i..t_f) = (s \in \mathsf{int}(t_i, t_f))$. The case $\tau^{-1}(s = t)$ is mapped to $(s \text{ is } t)$ only if s is either a variable or an integer constant, and to $(s = t)$ otherwise. It is also possible to extend the function τ to cover some other cases; for example, a CLP($\mathcal{SET}$)-constraint such as $s \in \{t_1, \dots, t_n\}$ where $t_1, \dots, t_n$ are integer constants, $\{t_1, \dots, t_n\} = i_1 \cup \dots \cup i_k$ and $i_1, \dots, i_k$ are intervals, can be translated as $\mathtt{s\ in\ i_1 \backslash/ \cdots \backslash/ i_k}$ (SICStus Prolog's syntax).

Proposition 1. *Let C be a $\mathcal{SET}$-constraint consisting of a conjunction of literals of the form $\ell \in \mathsf{int}(t_i, t_f), \ell = r, \ell \neq r$, where t_i, t_f are integer constants and ℓ and r are integer constants or variables. Then $\mathcal{FD} \models \exists\tau(C)$ iff $\mathcal{SET} \models \exists C$.*

Proof. (Sketch) Since $\mathcal{SET}$ contains a submodel isomorphic to $\mathcal{FD}$, the $(\rightarrow)$ direction follows trivially. For the $(\leftarrow)$ direction, if σ is s.t. $\mathcal{SET} \models \sigma(C)$, we build σ' s.t. $\mathcal{FD} \models \sigma'(\tau(C))$. If $\sigma(X)$ is not an integer constant, assign $\sigma'(X)$ to a large, unused integer. If $\sigma(X) = \sigma(Y)$, use the same integer for them. □

From a *practical* point of view, we know that, given a constraint C, $SAT_{\mathcal{SET}}(C)$ always detects its satisfiability or unsatisfiability. Conversely, the solvers for CLP($\mathcal{FD}$) are typically incomplete. We can obtain a complete solver by using labeling primitives. Specifically, if C is a $\mathcal{SET}$-constraint as in Proposition 1 and $\mathcal{V}_\in$ is the set of variables occurring in the $\in$-constraints, if $\mathcal{V}_\in = \mathit{vars}(C)$, then the CLP($\mathcal{FD}$) solver of SICStus Prolog returns a ground solution (or detect unsatisfiability) to the constraint $\tau(C) \wedge \mathtt{labeling}([\mathit{vars}(C)])$.

5.1 Integration of $SAT_{\mathcal{SET}}$ and $SAT_{\mathcal{FD}}$

We decompose each CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint in three parts: $C^S \wedge C^F \wedge C^C$, where C^S is a constraint that can be handled only by a $\mathcal{SET}$ solver, C^F is a constraint that can be handled only by a $\mathcal{FD}$ solver, and C^C is a constraint that can be handled by both solvers—i.e., C^C is composed of those constraints for which τ produces a result different from true. The C^C part effectively represents a communication conduit between the two constraint solvers. From now on we will replace each constraint with a pair $\langle C, D\rangle$, where $C = C^S \wedge C^C$ and $D = C^F \wedge \tau(C^C)$. The procedure $SAT_{\mathcal{SET}}$ is modified to handle a constraint subdivided into these two components, and to distribute the different components to the proper solver.

The modified rewriting rules for membership constraints are illustrated in the figure in the next page. Whenever a domain constraint $X \in \mathsf{int}(t_i, t_f)$ is encountered in C, the solver passes the corresponding $\mathcal{FD}$-constraint $\tau(c)$ to the $SAT_{\mathcal{FD}}$ procedure, which in turn will check the satisfiability of the constraint $\tau(c) \wedge D$. If $\tau(c) \wedge D$ turns out to be unsatisfiable, then also $SAT_{\mathcal{SET}}$ will terminate with a false result, otherwise, $SAT_{\mathcal{SET}}$ will continue. Note that $SAT_{\mathcal{FD}}$ may return an improved constraint D' (e.g., by narrowing intervals through node and arc consistency); in this case $SAT_{\mathcal{SET}}$ will be reactivated on $\langle C \wedge \tau^{-1}(D'), D'\rangle$. Moreover, the component $\tau^{-1}(D')$ returned to the $\mathcal{SET}$ solver may contain different types of constraints—i.e., simplified domain, equality and inequality constraints.

(1)	$\langle s \in \emptyset \wedge C', D\rangle$	$\mapsto$ false
(2)	$\langle r \in \{s \,\vert\, t\} \wedge C', D\rangle$	$\mapsto$ $\langle C' \wedge r = s, D \wedge \tau(r = s)\rangle$ *or* $\langle C' \wedge r \in t, D \wedge \tau(r \in t)\rangle$
(3)	$\langle t \in X \wedge C', D\rangle$	$\mapsto \langle X = \{t \,\vert\, N\} \wedge \mathsf{set}(N) \wedge C', D\rangle$
(4.1)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ *r is neither a variable nor an integer* t_i, t_f *integers*	$\mapsto$ false
(4.2)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = \mathsf{false}$	$\mapsto$ false
(4.3)	$\langle r \in \mathsf{int}(t_i, t_f) \wedge C', D\rangle$ t_i, t_f *integers* $SAT_{\mathcal{FD}}(D \wedge \tau(r \in \mathsf{int}(t_i, t_f))) = D'$	$\mapsto \langle C' \wedge \tau^{-1}(D'), D'\rangle$

Similar modifications can be applied to the procedures that handle the other primitive constraints, that we omit due to lack of space. Observe that each new constraint that is generated during $SAT_{\mathcal{SET}}$ computation should be translated with τ and made available to $SAT_{\mathcal{FD}}$ in the second part of the constraint. For example, each time we process constraints of the type $X = i$ with i integer constant, then the equation $X = i$ has to be provided to CLP($\mathcal{FD}$) as well. Observe, moreover, that $\mathcal{FD}$-constraints could be simply accumulated in the CLP($\mathcal{FD}$) constraint store, without solving them during the $SAT_{\mathcal{SET}}$ computation. The CLP($\mathcal{FD}$) constraint store could be processed by $SAT_{\mathcal{FD}}$ at the end, after the termination of $SAT_{\mathcal{SET}}$. Interleaving $SAT_{\mathcal{FD}}$ and $SAT_{\mathcal{SET}}$, as done in the rewriting rules discussed above, provides significant pruning of the search space.

5.2 The Global Constraint Solver

The global constraint solver $SAT_{\mathcal{SET}+\mathcal{FD}}$ is the same CLP($\mathcal{SET}$) constraint solver $SAT_{\mathcal{SET}}$ [5], in which CLP($\mathcal{SET}$) rewriting rules are replaced by the new rules defined in the previous section. These rules allow us to exploit the $SAT_{\mathcal{FD}}$ constraint solver whenever it is possible (and convenient).

$SAT_{\mathcal{SET}+\mathcal{FD}}(\langle C, D\rangle)$:
  set_infer(C);
  repeat
    $\langle C', D'\rangle := \langle C, D\rangle$;
    set_check(C);
    STEP($\langle C, D\rangle$);
  until $\langle C, D\rangle = \langle C', D'\rangle$

$SAT_{\mathcal{SET}+\mathcal{FD}}$ uses the procedure set_check to check the set constraints and the procedure set_infer to add set constraints for those variables that are required to belong to the sort Set; set_check and set_infer are the same procedures already used in CLP($\mathcal{SET}$), suitably updated in order to account for interval terms. The STEP procedure is the core part of $SAT_{\mathcal{SET}+\mathcal{FD}}$: it calls the rewriting procedures on the constraint $\langle C, D\rangle$ in a predetermined order (to ensure termination—see [5]). At the end of the execution of the STEP procedure, non-solved primitive constraints may still occur in $\langle C, D\rangle$; therefore, the execution of STEP has to be iterated until a fixed-point is reached—i.e., the constraint cannot be simplified any further.

As mentioned in Section 2, if C is satisfiable, then $SAT_{\mathcal{SET}}$ non-deterministically returns satisfiable constraints in solved form. This is not the case for the $SAT_{\mathcal{SET}+\mathcal{FD}}$

solver. Let $\langle C', D' \rangle$ be one of the pairs returned by the procedure $SAT_{\mathcal{SET}+\mathcal{FD}}$. $C' \wedge \tau^{-1}(D')$ is no longer guaranteed to be satisfiable in $\mathcal{SET}$. For example, a possible unsatisfiable output for $SAT_{\mathcal{SET}+\mathcal{FD}}$ is:

$\big\langle \mathtt{X} \notin \mathtt{S} \ , \mathtt{X} \neq \mathtt{Y} \wedge \mathtt{X} \neq \mathtt{Z} \wedge \mathtt{Y} \neq \mathtt{Z} \wedge \mathtt{X} \in 1..2 \wedge \mathtt{Y} \in 1..2 \wedge \mathtt{Z} \in 1..2 \big\rangle$.

On the other hand, observe that $SAT_{\mathcal{SET}+\mathcal{FD}}$ still preserves the set of solutions of the original constraint, thanks to Proposition 1. A complete solver can be obtained by adding chronological backtracking to force exploration of the solution space, e.g., by using labeling predicates. However, requiring direct execution of a complete labeling within $SAT_{\mathcal{SET}+\mathcal{FD}}$ each time $SAT_{\mathcal{FD}}$ is called would cause, in practice, the loss of most of the advantages of using the $SAT_{\mathcal{FD}}$ solver—it will effectively lead to a translation of each interval into an extensional set. Also, our approach is to maintain a behavior as close as possible to that of the traditional $SAT_{\mathcal{SET}}$ solver; thus we do not want to require the user to explicitly request an enumeration of the solutions.

The solution we propose is to modify the global solver as follows. Given a CLP$(\mathcal{SET})^{\mathsf{int}}$ constraint C, the resolution of C is performed according to the following steps:

step 1. The $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is initially applied to the constraint $\langle C, \mathsf{true} \rangle$; this will lead either to false (in this case we stop) or to the constraint $\langle C', D' \rangle$.

step 2. Let D' be the $\mathcal{FD}$ part of the constraint obtained from the last call to $SAT_{\mathcal{SET}+\mathcal{FD}}$. A constraint of the form $X \in t_1..t_2$ is selected from D' and the solutions $R_1, \ldots, R_k$ are collected from the execution of $SAT_{\mathcal{FD}}(D' \wedge \mathsf{labeling}([X]))$. Each R_i, in particular, will contain a constraint of the type $X = r_i$ for some $r_i \in t_1..t_2$. The execution of $SAT_{\mathcal{SET}+\mathcal{FD}}$ is restarted from each constraint $\langle C' \wedge \tau^{-1}(R_i), R_i \rangle$. The step is repeated until there are no constraints of the form $X \in t_1..t_2$ in D'.

Additional improvements to the resolution process can be achieved through a refinement of the sorts system used by CLP$(\mathcal{SET})^{\mathsf{int}}$. Currently the system recognizes two sorts, `Set` and `Ker`, describing respectively set terms and non-set terms. The sort `Ker` requires now a further refinement, by distinguishing integer constants (`Int`) from all other non-set terms (`Ker`). Enforcing the sorts requires the introduction of additional constraints: $\mathsf{integer}(s)$ and $\mathsf{ker}(s)$ are satisfiable only by terms belonging to the sort `Int` and `Ker`, respectively. During the rewriting process, each time a constraint of the type $s \in \mathsf{int}(t_i, t_f)$ is generated, the following constraint is introduced: $\mathsf{integer}(s) \wedge \mathsf{integer}(t_i) \wedge \mathsf{integer}(t_f)$. Observe that if one of the endpoints of the interval (i.e., t_i or t_f) is a variable, then the new constraint will guarantee that the variable can be assigned only integer values. Constraints $\mathsf{integer}(X)$ are also generated via static program analysis, as shown in Section 7.

6 Encoding New Operators

CLP$(\mathcal{FD})$ allows us to properly and efficiently deal with constraints of the form $\ell \in t_i..t_f, \ell = r, \ell \neq r, \ell < r, \ell \leq r$. Moreover, arithmetic operations can be used in terms ℓ and r. The capability of dealing with arithmetic operations and with the natural ordering between integers can be used to improve the capabilities of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ as well. We could deal with this new constraints directly in CLP($\mathcal{SET}$), since with a set-based encoding of integer terms, $<$ can be directly encoded by $\in$. However, this approach is highly inefficient, and, moreover, it turns out to be not adequate to deal with more general algebraic expressions like $X = Y + 3 \wedge X - Z \leq Y + 1$, since arithmetic function symbols are not interpreted in CLP($\mathcal{SET}$).

A more effective solution can be achieved by avoiding any direct treatment of $<$ and $\leq$ within $SAT_{\mathcal{SET}}$, and simply relying on $SAT_{\mathcal{FD}}$ to handle this type of constraints. Let us assume that the language of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ is extended as follows: $\Pi_c = \{=, \in, \cup_3, \|, \mathsf{set}, \mathsf{integer}, \leq\}$ and that $\mathcal{F}$ includes the traditional arithmetic function symbols (e.g., $+$, $-$). Let us call *integer terms* those terms built using arithmetic operators, integer constants and variables. Integer terms are interpreted on the subset of the domain of $\mathcal{SET}$ containing integer constants (i.e., on terms of sort `Int`). Interpretation of arithmetic function symbols is the expected one. Their sort will be $\langle\{\mathtt{Int}\}, \{\mathtt{Int}\}, \{\mathtt{Int}\}\rangle$.

Integer terms can occur as arguments of primitive constraints based on $=$, $\neq$, and $\leq$. Rewriting rules for these constraints are modified so that as soon as they recognize that one of the arguments of the predicate symbol at hand is an integer term, solution of the constraint is committed to $SAT_{\mathcal{FD}}$. The translation function τ is modified to allow s, t_i, t_f, t to be either variables or integer terms (not only constants), and such that $\tau(s \leq t) = s \leq t$, where s, t are either integer terms or variables. The rewriting rules for $\leq$ constraints used in $SAT_{\mathcal{SET}+\mathcal{FD}}$ are defined as follows:

(1)	$\langle s \leq t \wedge C, D\rangle$ *s or t not integers or* $SAT_{\mathcal{FD}}(D \wedge \tau(s \leq t)) = \mathsf{false}$	$\mapsto \mathsf{false}$
(2)	$\langle s \leq t \wedge C, D\rangle$ *s, t integers or variables* $SAT_{\mathcal{FD}}(D \wedge \tau(s \leq t)) = D'$	$\mapsto \langle C \wedge \tau^{-1}(D'), D'\rangle$

The definitions for the $=$ and $\neq$ constraints are updated accordingly. We assume that terms built using arithmetic operators but involving also non-integer terms are considered ill-formed terms (i.e., not respecting the sorts). For instance $X + f(a)$ is an ill-formed term. In CLP($\mathcal{SET}$)$^{\mathsf{int}}$ the presence of ill-formed syntactic objects is detected within the constraint solver procedure by exploiting sort constraints, set, $\mathsf{integer}$, and ker, automatically generated by the solver itself or provided by the user as part of the input constraint. Whenever the solver detects an inconsistency among sort constraints it terminates with a false result.

The presence of $<$ and $\leq$ allows us also to introduce new expressive constraints in $\mathcal{SET}$. For example, it becomes possible to provide a general cardinality constraint size in the context of CLP($\mathcal{SET}$): the constraint $\mathsf{size}(s, t)$ is satisfied if s is a set, t is an integer, and t corresponds to the cardinality of s. These constraints can be handled using new rewriting rules that we omit due to lack of space.

As an example of how constraints on integer terms can be conveniently exploited in CLP($\mathcal{SET}$)$^{\mathsf{int}}$ consider the problem of finding solutions for a system of linear equations. The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ implementation for this problem, while the table shows experimental results obtained running the code using $SAT_{\mathcal{SET}}$ and using $SAT_{\mathcal{SET}+\mathcal{FD}}$.

$$
\begin{array}{l}
\texttt{start}(X, Y, Z, L, H) :- \\
\quad X \in \mathsf{int}(L, H) \wedge \\
\quad Y \in \mathsf{int}(L, H) \wedge \\
\quad Z \in \mathsf{int}(L, H) \wedge \\
\quad X1 = 1 + X \wedge \\
\quad X1 = 2 * Y + Z \wedge \\
\quad X2 = Z - Y \wedge X2 = 3 \wedge \\
\quad X3 = X + Y \wedge X3 = 5 + Z.
\end{array}
$$

	Experimental results						
L	-10	-20	-30	-10^3	-10^4	-10^5	-10^6
H	10	20	30	10^3	10^4	10^5	10^6
$SAT_{\mathcal{SET}}$	41s	8m 31s	38m	∞	∞	∞	∞
$SAT_{\mathcal{SET}+\mathcal{FD}}$	0s	0s	0s	0.13s	1.6s	19.5s	3m 58s

Solving the problem using $SAT_{\mathcal{SET}}$ is feasible but with unacceptable computation time. Conversely, the solution using the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is obtained in time comparable to that obtained using directly CLP($\mathcal{FD}$).

As another example, consider the well-known combinatorial problem of solving the SEND + MORE = MONEY puzzle (i.e., solve the equation by assigning a distinct digit between 0 and 9 to each letter). The following code shows a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ possible solution for this problem.

$$
\begin{array}{l}
\texttt{solve\_puzzle}(S, E, N, D, M, O, R, Y) :- \\
\qquad \{S, E, N, D, M, O, R, Y\} \subseteq \mathsf{int}(0, 9) \ \wedge \\
\qquad \texttt{size}(\{S, E, N, D, M, O, R, Y\}) = 8 \ \wedge \\
\qquad M * 10000 + O * 1000 + N * 100 + E * 10 + Y \ = \\
\qquad\qquad\qquad S * 1000 + E * 100 + N * 10 + D \ + \\
\qquad\qquad\qquad M * 1000 + O * 100 + R * 10 + E.
\end{array}
$$

Running the program using $SAT_{\mathcal{SET}}$ results in unacceptable computational time, while using $SAT_{\mathcal{SET}+\mathcal{FD}}$ we get the solution in 0.01 seconds.

As a final remark, observe that effective solutions have been proposed to handle *cardinality* constraints in CLP($\mathcal{FD}$) [16]. This concept has a semantics related to the size of a set, but it is not the general cardinality notion. Precisely, given n variables $X_1, \ldots, X_n$ with domains $D_1, \ldots, D_n$, a cardinality constraint for a domain element d is a bound k on the number of X_i that can be simoultaneously instantiated to d. It is therefore a constraint associated to the availability of a fixed number k of resources of a certain kind d.

7 Static Analysis for Nested Sets Identification

It is possible to further improve communication between the two constraint solvers by *statically* extracting selected properties from programs. We introduce a static analysis schema for CLP($\mathcal{SET}$)$^{\mathsf{int}}$ programs, checking if the term s is a *n*ested *set* of *i*ntegers. We define $\mathsf{ni\_set}(s)$ as the measure of the depth of nesting of the sets. The intuitive semantics is the following:

- $\mathsf{ni\_set}(s) = 0$ if s in an integer constant;
- $\mathsf{ni\_set}(s) = n + 1$ if s is a set and for each $t \in s$ it holds that $\mathsf{ni\_set}(t) = n$.

(1)	$\emptyset$	$\mapsto \mathsf{ni\_set}(\emptyset) := \bot$
(2)	$f(t_1,\dots,t_k)$, $f \not\equiv \{\cdot \mid \cdot\}, f \not\equiv \mathsf{int}$, *not an integer constant*	$\mapsto \mathsf{ni\_set}(f(t_1,\dots,t_k)) := \top$
(3)	*an integer constant* t	$\mapsto \mathsf{ni\_set}(t) := 0$
(4)	$\mathsf{int}(t_i,t_f)$, t_i,t_f *are integers and* $t_i \leq t_f$	$\mapsto \mathsf{ni\_set}(\mathsf{int}(t_i,t_f)) := 1$
(5)	$\{t \mid s\}$	$\mapsto$ $\mathsf{ni\_set}(\{t \mid s\}) := (\mathsf{ni\_set}(t)+1) \sqcup \mathsf{ni\_set}(s)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(\{t \mid s\}) - 1$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(\{t \mid s\})$
(6)	$s = t$	$\mapsto$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(s)$
(7)	$t \in s$	$\mapsto$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(t) \sqcup (\mathsf{ni\_set}(s)-1)$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup (\mathsf{ni\_set}(t)+1)$
(8)	$\cup_3(r,s,t)$	$\mapsto$ $\mathsf{ni\_set}(r) := \mathsf{ni\_set}(r) \sqcup \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t)$; $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(r)$; $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(r)$

Fig. 1. Abstract Interpretation steps to compute ni_set

Observe that $\mathsf{ni\_set}(s) > 0$ implies that the constraint $\mathsf{set}(s)$ holds. If $\mathsf{ni\_set}(s) = 1$ then s is a flat set of integers (e.g., $\{1,2,3,5\}$). Moreover, we allow two additional possible values for $\mathsf{ni\_set}(s)$: $\bot$, when no information is currently available for s, and $\top$, when s is not a nested set of integers. The values for $\mathsf{ni\_set}(s)$ are elements of the lattice $\mathcal{L} = \langle\{\bot,\top,0,1,2,\dots\},\prec\rangle$ represented in Figure 8. The join operator $\sqcup$ is naturally defined on $\mathcal{L}$. We introduce two more operations on the lattice $\mathcal{L}$, respectively denoted by "+" and "−", defined in the table below, where m,n,p denote integers, $+_{\mathbb{Z}}$ and $-_{\mathbb{Z}}$ represent the standard addition and subtraction between integers, and x is a generic element of $\mathcal{L}$:

$\bot+\bot=\bot,\ \bot+n=n+\bot=\bot,\ \top+x=x+\top=\top,\ m+n=m+_{\mathbb{Z}}n$
$\top-x=\top, m-n=m-_{\mathbb{Z}}n\ \ (m\geq n), m-n=\top\ \ (m<n),$ $\bot-\bot=\bot-n=\bot,\bot-\top=\top$

The process of deriving the values of $\mathsf{ni\_set}(\cdot)$ starts by initializing $\mathsf{ni\_set}(X)$ to $\bot$ for each variable X occurring in the CLP($\mathcal{SET}$)$^{\mathsf{int}}$ goal G under consideration. The rules in Figure 1 are then applied on the terms and subterms of G and constraint literals until a fixpoint is reached. The case $s||t$ is missing since no information concerning the elements of s and t can be obtained by the fact that they are disjoint.

It is immediate to see that the analysis is correct and that terminates in polynomial time w.r.t. the size of G. It can be further refined, by extracting additional information from the arithmetic operations. For example, from $A+B$ one can infer that $\mathsf{ni\_set}(A) = \mathsf{ni\_set}(A) \sqcup 0$ (the same for B). Similarly, the presence of interval definitions with variables as endpoints, as in $X \in \mathsf{ni\_set}(1,N)$ allows us to infer: $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(N) \sqcup 0$.

For example, consider the constraint: $\mathtt{M} \in \mathsf{int}(1,10), \mathtt{N} = \{\{\mathtt{M}\},\mathtt{M}\}, \cup_3(\{\mathtt{M}\},\emptyset,\mathtt{S})$. The analysis returns $\mathsf{ni\_set}(M) = 0, \mathsf{ni\_set}(S) = 1, \mathsf{ni\_set}(N) = \top$.

8 Analysis of CLP($\mathcal{SET}$)$^{\mathsf{int}}$ Programs

The set of rules presented in the previous section can be used to analyze a complete CLP($\mathcal{SET}$)$^{\mathsf{int}}$ program. Let us assume w.l.o.g. that a program P is composed of a set of clauses of the form $p(X_1,\dots,X_m) \mathrel{:-} C, B$ where $X_1,\dots,X_m$

are distinct variables, C is a CLP($\mathcal{SET}$)$^{\mathsf{int}}$ constraint, and B is a conjunction of program atoms of the form $q(Y_1, \ldots, Y_k)$, where $Y_1, \ldots, Y_k$ are distinct variables.

During the analysis, each variable is assigned an abstract value that represents its properties. Since the variables are managed with a global scope, we avoid possible name clashes that can affect the analysis, renaming the variables in such a way that each of them occurs only in one clause. We also force certain collisions that may help the analysis task. Consider the set of n clauses defining p: the variables are renamed such that all these clauses have identical heads. Thus, given $p(X_1, \ldots, X_k) \coloneq C_1, B_1, \ldots, p(Y_1, \ldots, Y_k) \coloneq C_n, B_n$, they are renamed to $p(N_1, \ldots, N_k) \coloneq C_1[X_i/N_i], B_1[X_i/N_i] \ldots p(N_1, \ldots, N_k) \coloneq C_n[Y_i/N_i], B_n[Y_i/N_i]$, where N_i are new variables. The analysis gathers more information if we consider the program together with its input . For instance, assume that the goal G contains the m variables $X_1, \ldots, X_m$ and that $X_1, \ldots, X_n$ $(n < m)$ are associated to integer numbers. Then, choosing some integer numbers $s_1, \ldots, s_n$, we can make available the information to the analysis by adding to the program P the new clause $g(X_1, \ldots, X_n, X_{n+1}, \ldots, X_m) \coloneq X_1 = s_1 \wedge \ldots \wedge X_n = s_n, G$.

The analysis algorithm proceeds as follows. At the beginning, all the program variables are initialized with $\perp$ from the lattice $\mathcal{L}$. The program analysis performs the fixpoint described in the previous section to determine the $\mathsf{ni\_set}$ values, by repeatedly executing the following two steps for each clause in the program:

- *Local analysis:* The analysis described in Section 7 is applied to C.
- *Propagation of information to other clauses:* For each $q(A_1, \ldots, A_k)$ in B, the information related to formal (X_i) and actual (A_i) parameters is updated as follows: $\mathsf{ni\_set}(X_i) := \mathsf{ni\_set}(X_i) \sqcup \mathsf{ni\_set}(A_i)$; $\quad \mathsf{ni\_set}(A_i) := \mathsf{ni\_set}(X_i)$;

Domain Analysis. The static analysis scheme can be also used to infer domain information for those variables X such that $\mathsf{ni\_set}(X) = 0$. In this case, the elements of the abstraction lattice $\mathcal{D}$ are integer intervals. The order in the lattice is provided by the relation $[x, y] \leq [x', y']$ iff $x' \leq x$ and $y \leq y'$. The join operation $\sqcup$ of two intervals is: $[x, y] \sqcup [x', y']$=$[\min(x, x'), \max(y, y')]$. The analysis rules are similar to those in Figure 1. As an example, consider the CLP($\mathcal{SET}$) program:

$a(X, Y) \coloneq X \in \mathsf{int}(1, 3) \wedge X \neq Y \wedge p(Y). \quad p(Z) \coloneq Z = 3. \quad p(Z) \coloneq Z = 5.$
The analysis determines the following properties: $\mathsf{ni\_set}(X) = 0, \mathsf{ni\_set}(Y) = 0, \mathsf{ni\_set}(Z) = 0, X \in \mathsf{int}(1, 3), Y \in \mathsf{int}(3, 5), Z \in \mathsf{int}(3, 5)$. Without this information, a constraint such as $\mathtt{X} \neq \mathtt{Y}$ would be maintained in the $\mathcal{SET}$ part of the constraint, while, after the analysis, it is possible to pass it to $SAT_{\mathcal{FD}}$, and to use X and Y as arguments of a $\mathsf{labeling}$. Thus, this analysis helps in preparing the domain constraints to be sent to CLP($\mathcal{FD}$).

Application of the Static Analysis Framework. The results of the static analysis can generate new constraints. For all variables X such that $\mathsf{ni\_set}(X) = 0$ we add the constraints $\mathsf{integer}(X)$ and $X \in \mathtt{dom}(X)$.

Moreover, in some cases it is possible to directly compile a fragment of a CLP($\mathcal{SET}$) program P into CLP($\mathcal{FD}$). This can be done, for example, when for all term s in P, $\mathsf{ni\_set}(s) \in \{0, 1\}$. Consider the following program, used to find subsets S of $\{1,\ldots,n\}$ such that, for each pair of (not necessarily distinct) numbers in S, their sum is not in S (simplification of Shur numbers problem):

$$\begin{array}{l} shur(N,S) :- \\ S \subseteq \mathsf{int}(1,N) \ \wedge \\ (\forall X \in S)(\forall Y \in S)(\forall Z \in S)(Z \neq X+Y). \end{array}$$

The $\forall$ are used to encode RUQs (see Section 2) . The static analysis process is applied to the program obtained from the removal of the RUQs, and produces $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(X) = \mathsf{ni\_set}(Z) = \mathsf{ni\_set}(T) = 0, \mathsf{ni\_set}(S) = 1$. These results can be used to directly derive a SICStus CLP($\mathcal{FD}$) program. The two programs have been executed for different values of N, using a PC, 1GHz, using SICStus Prolog 3.10. Precisely, in {log} (implementation of CLP($\mathcal{SET}$) [12]) the goal is `Q={A:shur(N,A)}` and in SICStus (execution of the program automatically derived from CLP($\mathcal{SET}$)$^{\mathsf{int}}$ using static analysis) `setof(A,shur(N,A),Q)`. In Figure 8 we report the running times.

N	5	6	7	8	16	24
{log}	11.7s	54s	3m.57s	∞	∞	∞
CLP($\mathcal{FD}$)	0.01s	0.02s	0.04s	0.08s	6.5s	3m.48s
Speedup	1200	2700	6000	∞	∞	∞

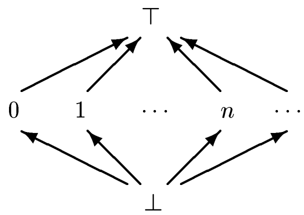

Fig. 2. Computational results and the Lattice $\mathcal{L}$

9 Conclusions

In this paper we presented an extension of the constraint logic programming language on hereditarily finite sets CLP($\mathcal{SET}$). The extension provides the ability to combine manipulation of domain variables (as in CLP($\mathcal{FD}$)) with the ability to manipulate the domain themselves, seen as finite sets. On one hand, this extends the expressive power of CLP($\mathcal{FD}$), allowing domains to play the role of first-class citizens of the language. On the other hand, the ability to automatically determine domain/domain variables allows the system to effectively distribute constraints between two constraints solvers (the one for CLP($\mathcal{SET}$) and the one for CLP($\mathcal{FD}$)), thus using the most efficient resolution mechanisms to solve each constraint. Static analysis mechanisms have been developed to support the execution of the extended constraint solving system. The extensions described in this paper have been implemented and will be soon available in the current distribution of the {log} system [12].

The work presented provides a number of new avenues for further generalizations. We plan to extend the results to sets of integer (or elements chosen from a finite universe) elements and to sets of sets of integer elements. These situations are precisely identified by the program analysis technique, when the outputs for a set s are $\mathsf{ni\_set}(s) = 1$ and $\mathsf{ni\_set}(s) = 2$, respectively. Several algorithms are based on operations on these two kinds of sets. For instance graphs related problems deal with sets of nodes and sets of sets of nodes (edges and/or partitions). Similarly, cooperations with constraint solvers on real numbers, as in $CLP(\mathcal{R})$, will be investigated.

Acknowledgements. We thank Sara Manzoli for useful discussions. The work is partially supported by MIUR project: *Automatic Aggregate—and number—Reasoning for Computing.*

References

1. P. Arenas-Sánchez and M. Rodríguez-Artalejo. A General Framework for Lazy Functional Logic, Programming with Algebraic Polymorphic Types. *Theory and Practice of Logic Programming*, 2(1):185–245, 2001.
2. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
3. P. Codognet and D. Diaz. Compiling constraints in CLP(FD). *J. of Logic Programming*, 27:185–226, 1996.
4. A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *J. of Logic Programming*, 28(1):1–44, 1996.
5. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *ACM TOPLAS*, 22(5):861–931, 2000.
6. P. Flener, B. Hnich, and Z. Hiziltan. Compiling high-level type constructor in constraint programming. In I.V. Ramakrishnan, editor, *PADL2001*, LNCS 1990, pages 229–244. Springer-Verlag, Berlin, 2001.
7. C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1:191–246, 1997.
8. IC Parc. *The ECLiPSe Constraint Logic Programming System*. London. `www.icparc.ic.ac.uk/eclipse/`.
9. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19–20:503–581, 1994.
10. B. Jayaraman. Implementation of Subset-Equational Programs. *J. of Logic Programming*, 12(4):299–324, 1992.
11. K. Marriott and P. J. Stuckey. *Programming with Constraints: an Introduction*. The MIT Press, Cambridge, Mass., 1998.
12. G. Rossi. *The {log} Constraint Logic Programming Language*. `http://prmat.math.unipr.it/~gianfr/setlog.Home.html`.
13. O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of Set Terms in the Logic Data Language (LDL). *J. of Logic Programming*, 12(1/2):89–119, 1992.
14. Swedish Institute of Computer Science. *The SICStus Prolog Home Page*. `www.sics.se`.
15. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
16. P. Van Hentenryck and Y. Deville. The Cardinality Operator: a New Logical Connective for Constraint Logic Programming. In K. Furukawa, editor, *International Conference on Logic Programming*, pages 745–759. MIT Press, 1991.
17. T. Yakhno and E. Petrov. Extensional Set Library for ECLiPSe. In D. Bjørner, M. Broj, and A. Zamulin, editors, *PSI'99*, LNCS 1755, pages 434–444. Springer-Verlag, Berlin, 2000.