A Constraint Logic Programming Framework for Effective Programming with Sets and Finite Domains*

A. Dovier
Università di Udine
Dip. di Mat. e Informatica
dovier@dimi.uniud.it

E. Pontelli
New Mexico State University
Dept. Computer Science
epontell@cs.nmsu.edu

A. Dal Palù and G. Rossi
Università di Parma
Dip. di Matematica
alessandro.dalpalu@unipr.it
gianfranco.rossi@unipr.it

February 25, 2006

Abstract

In this paper we propose a new Constraint Logic Programming language that integrates the languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). The language presented, called CLP($\mathcal{SET}, \mathcal{FD}$) is an extension of both CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$) and it allows 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. We describe an implementation of the resulting system and present a number of benchmark results.
Keywords. Programming with Sets, Constraint Logic Programming, Integration of Languages

1 Introduction

The literature is rich of proposals aimed at developing declarative programming frameworks that incorporate different types of *set-based primitives* (e.g., [2, 20, 26, 10, 12, 5]). These frameworks provide a higher level of abstraction and declarativity, thanks to the direct availability of the popular notation and operations of *set theory*. These features make this type of languages particularly suited for problem modeling and rapid software prototyping—in fact, many high-level software modeling languages (e.g., [1, 9]) provide set abstractions as first-class citizens of the language.

Among these many proposals in the field of programming with sets, CLP($\mathcal{SET}$) [12] provides very flexible and general forms of sets while preserving a clean semantics and a true declarative reading. CLP($\mathcal{SET}$) is a constraint logic programming language whose constraint domain is that of *hereditarily finite sets*—i.e., finitely nested sets that are finite at each level of nesting. CLP($\mathcal{SET}$) allows sets to be *nested* and *partially specified*—e.g., set elements can contain unbound variables and it is possible to operate with sets that have been only partially specified. CLP($\mathcal{SET}$) provides a collection of primitive constraint predicates adequate to

*A preliminary version of this paper, titled *"Integrating Finite Domain Constraints and CLP with Sets"*, appeared in the proceedings of the International Conference on Principles and Practice of Declarative Programming, ACM Press, pp. 219–229, 2003. **Contact Author:** Enrico Pontelli, Dept. Computer Science, New Mexico State University, Box 30001/CS, Las Cruces, NM 88003, USA, epontell@cs.nmsu.edu represent all the most commonly used set-theoretic operations (e.g., union, intersection, difference). In [12], a complete constraint solver for this language is described, capable of deciding the satisfiability of arbitrary conjunctions of these primitive set constraints.

One of the main criticisms attributed to many of the approaches presented in the literature dealing with computations in presences of set abstractions is that the intrinsic computational costs of some set operations—e.g., *unification between sets*, which is a known NP-complete problem—make these languages suited for modeling and fast prototyping, but inadequate for the direct development of practically usable solutions.

On the other hand, various proposals have been made to introduce efficient constraint-based language primitives for dealing with *specialized* classes of sets—e.g., *finite* subsets of a predefined *domain* (typically, the set $\mathbb{Z}$ of integer numbers). Constraint-based manipulation of finite domains provides the foundations of the popular CLP($\mathcal{FD}$) framework [6, 7, 28], that has been demonstrated well-suited for the encoding and the fast resolution of combinatorial and optimization problems [15, 22].

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 arbitrary hereditarily finite sets—as possible in CLP($\mathcal{SET}$)—and the efficient use of constraint propagation techniques provided by constraint solving over finite domains—as possible in CLP($\mathcal{FD}$). The combination of the two models leads to a framework, called CLP($\mathcal{SET}, \mathcal{FD}$), where high-level set operations allow flexible manipulation of domains (viewed as sets), while the constraint propagation mechanisms of CLP($\mathcal{FD}$) allow the efficient handling of simple sets whenever feasible.

Example 1 *Consider the simple constraint*

$$X \in \{1, 2, 3\}, \ Y \in \{3, 2\}, \ X > Y.$$

The predicate '$>$' is not a constraint predicate symbol in CLP($\mathcal{SET}$) and, as in Prolog, it requires two ground arguments for its correct evaluation. Thus, the above constraint, when processed in CLP($\mathcal{SET}$), requires the explicit enumeration of the possible bindings of X and Y to detect the unique solution $X = 3$ and $Y = 2$. On the other hand, bounds consistency employed by CLP($\mathcal{FD}$) allows one to directly jump to the desired solution, thus enhancing the efficiency of the CLP($\mathcal{SET}$) framework. □

Example 2 *Consider the following simple encoding of an instance of the graph coloring problem, encoded in CLP($\mathcal{FD}$):*

$$Dom = \{r, g, b\}, X \in Dom, Y \in Dom, Z \in Dom, X \neq Y, X \neq Z, Y \neq Z$$

where the variables represent nodes of a graph and the '$\neq$' constraints encode the edges of the graph. The same problem can be expressed in more declarative and compact way using the following CLP($\mathcal{SET}$)-constraint:

$$\{\{X, Y\}, \{X, Z\}, \{Y, Z\}\} \subseteq \{\{r, g\}, \{r, b\}, \{g, b\}\}$$

□

In order to accomplish the proposed integration, we propose a natural extension of the existing constraint handling capabilities of the CLP($\mathcal{SET}$) framework, by developing an intelligent cooperation between constraint solving on integer constants and intervals and constraint solving over hereditarily finite sets. In particular, intervals are naturally viewed as *set terms* and processed according to such view. We integrate the CLP($\mathcal{FD}$) constraint solver and the CLP($\mathcal{SET}$) constraint solver, by allowing the latter to exploit the former whenever constraints over intervals or equations and disequations over integers have to be processed. We develop novel constraint solving algorithms that allow the two 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, that allow us to automatically discover situations where the CLP($\mathcal{FD}$) constraint solver can be used in place of the CLP($\mathcal{SET}$) solver to check constraint satisfiability. The technique is based on the ability of the program analyzer to determine those finite sets of integers which can be easily mapped to intervals, and therefore are likely to be processed more efficiently by the CLP($\mathcal{FD}$) constraint solver. Moreover, the analyzer can detect entire fragments of CLP($\mathcal{SET}$) programs that can be completely translated to equivalent CLP($\mathcal{FD}$) programs and executed more efficiently.

1.1 Related Work

This work represents a natural evolution of our previous research on programming with sets, starting from the logic language {log} [10], and later developed as a constraint logic programming language (CLP($\mathcal{SET}$)) [12] 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 (in particular, CLP($\mathcal{SET}, \mathcal{FD}$) inherits from CLP($\mathcal{FD}$) the capability of handling integer constraints). The usefulness of languages with high-level domain manipulation capabilities has been highlighted by various authors, e.g., [14, 30].

A work similar in spirit to the one provided here is the proposal by Flener et al. [13], where the authors compile the language ESRA, that includes set-based primitives, into the modeling language OPL. Another related proposal is the one by Gervet [15], where some classes of set operations over finite domains of flat sets are developed in the context of the CLP language ECLiPSe [17]. In [32] the authors propose a tuple-based treatment of sets in ECLiPSe and compare it with Gervet's approach. Both these two approaches, however, do not allow a complete and general handling of programs involving sets.

1.2 Paper Organization

The rest of this paper is organized as follows. In Section 2, we summarize the main distinguishing features of the languages CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). Syntax and semantics of the language CLP($\mathcal{SET}, \mathcal{FD}$) are presented in Section 3. In Section 4, we describe the constraint solver for the new language, as integration of the two constraint solvers for CLP($\mathcal{SET}$) and CLP($\mathcal{FD}$). In Section 5, we show how to use static analysis to obtain information useful to the distribution of the constraints to the two constraint solvers. Experimental results are presented in Section 6. Conclusions and future work are reported in Section 7.

2 Background and Definitions

In this section, we review the basic definitions relative to the two instances of the CLP programming scheme that we propose to integrate. The presentation is used to introduce the basic notation and it does not have any pretense of completeness—the interested reader is referred to [18, 19, 12] for a complete treatment of these topics.

2.1 The CLP Scheme

Constraint Logic Programming (CLP) [18] is a logic programming paradigm which relies on the use of predicates and functions having a pre-defined interpretation over a fixed domain (the *constraint domain*). CLP is parametric w.r.t. a first-order language and constraint domain $\mathcal{X}$. The first-order signature Σ of $\mathcal{X}$ is composed of

- a set $\mathcal{F}$ of constant and function symbols,
- a set Π_C of constraint predicate symbols, containing the equality symbol $=$,
- a set Π_U of user-defined predicate symbols, and
- a denumerable set $\mathcal{V}$ of variables.

We will talk about $\langle \Pi, \mathcal{F}, \mathcal{V} \rangle$-atoms, $\langle \Pi, \mathcal{F}, \mathcal{V} \rangle$-literals, etc. to denote atoms, literals, etc. built using the symbols from the signature $\langle \Pi, \mathcal{F}, \mathcal{V} \rangle$.

A $\mathcal{X}$*-constraint* is a first-order formula of the signature $\Sigma_{\mathcal{X}} = \langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$. Not all first-order formulae of $\Sigma_{\mathcal{X}}$, in general, are considered $\mathcal{X}$-constraints: the class of *admissible* $\mathcal{X}$-constraints is identified by $\mathcal{C}_{\mathcal{X}}$. In the CLP languages presented in this paper, $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$-atoms are $\mathcal{X}$-constraints (also called *primitive constraints*), while $\mathcal{X}$-constraints are composed of all the conjunctions of primitive and negated primitive constraints.

$\mathcal{X}$-constraints are interpreted over a predefined domain. Let $vars(\gamma)$ denote the set of variables present in the formula γ. Let φ and ψ are two first-order formulae. Let $\vec{A} = vars(\varphi) \cap vars(\psi)^1$, $\vec{B} = vars(\varphi) \setminus vars(\psi)$, and $\vec{C} = vars(\psi) \setminus vars(\varphi)$. We say that φ and ψ are *equisatisfiable* w.r.t. $\mathcal{X}$ if

$$\mathcal{X} \models \forall \vec{A}\, (\exists \vec{B}\, (\varphi) \leftrightarrow \exists \vec{C}(\psi))$$

We will use the notation $\vec{\exists}\gamma$ as a short form for $\exists \bar{X}\gamma$, with $\bar{X} = vars(\gamma)$.

A *CLP*$(\mathcal{X})$ clause is of the form H :- $B_1, \ldots, B_k$, where H is a $\langle \Pi_U, \mathcal{F}, \mathcal{V}\rangle$-atom and $B_1, \ldots, B_k$ are either $\langle \Pi_U, \mathcal{F}, \mathcal{V}\rangle$-atoms or $\mathcal{X}$-constraints.

Constraint solvers are procedures that, given a constraint C, return a set $C_1, \ldots, C_k$ of constraints whose disjunction is equisatisfiable to C w.r.t. $\mathcal{X}$. A *complete constraint solver* is such that the returned constraints $C_1, \ldots, C_k$ are in a *canonical form*. Intuitively, a constraint C_i is in canonical form if a solution of C_i can be computed easily, and it is also natural to enumerate its solution sets. If C is unsatisfiable w.r.t. $\mathcal{X}$, then $k = 0$ (syntactically, it is returned false). An *incomplete constraint solver*, instead, typically returns only *one* (i.e., $k = 1$) constraint which has the same set of solutions as the original constraint C (i.e., it is equisatisfiable to C); determining the satisfiability of this resulting constraint can be potentially hard.

For a more in-depth discussion of the foundations of Constraint Logic Programming, the interested reader is referred to [18, 19].

2.2 CLP($\mathcal{SET}$)

In the case of CLP($\mathcal{SET}$) [12], the signature Σ is defined as follows:

- $\mathcal{F}$ includes the constant $\emptyset$ and the binary function symbol $\{\cdot \mid \cdot\}$;
- Π_C is the set $\{=, \in, \cup_3, ||, \mathsf{set}\}$;
- Π_U is the set of user-defined predicate symbols;
- $\mathcal{V}$ is a denumerable set of variables.

The intuitive semantics of the various symbols is the following: the symbol $\emptyset$ represents the empty set, while $\{\cdot \mid \cdot\}$ is used as a *set constructor*: $\{t \mid s\}$ represents the set composed of the elements of the set s plus the element t (i.e., $\{t \mid s\} = \{t\} \cup s$).

The predicates $=$ and $\in$ represent the equality and the membership relations. The predicate $\cup_3$ represents the union relation: $\cup_3(r, s, t)$ holds iff $t = r \cup s$. The predicate $||$ represents the disjoint relationship between two sets: $s \,||\, t$ holds iff $s \cap t = \emptyset$.

CLP($\mathcal{SET}$) is multi-sorted; terms are separated into two sorts, one (called Set) representing set terms, and one (called Ker) 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. We only consider *well-formed* terms; a well-formed term requires the second argument of a term built using the functor $\{\cdot \mid \cdot\}$ to be a term of sort Set.

We use the notation $\{t_1, t_2, \ldots, t_n \mid t\}$ as a shorthand 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 $\{t_1 \mid \{t_2 \mid \cdots \{t_n \mid \emptyset\} \cdots\}\}$. Observe that one can write terms representing sets which are nested at any level; for instance: $\{\, X, \{\emptyset, \{a\}\}, \{\{\{b\}\}\}\,\}$.

A $\mathcal{SET}$*-constraint* is a conjunction of $\langle \Pi_C, \mathcal{F}, \mathcal{V}\rangle$-literals. Other basic set-theoretic operations (e.g., $\cap$, $\subseteq$) are easily defined using $\mathcal{SET}$-constraints. CLP($\mathcal{SET}$) provides also a syntactic extension called *Restricted Universal Quantifiers (RUQs)*, i.e., 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 conjunction of $\langle \Pi_U, \mathcal{F}, \mathcal{V}\rangle$-atoms. The semantics of $(\forall X \in S)G$ is $\forall X\, (X \in S \rightarrow G)$. The implementation of RUQs is accomplished via program transformation—generating two recursive CLP($\mathcal{SET}$) clauses using set terms and $\mathcal{SET}$-constraints [10].

The semantics of CLP($\mathcal{SET}$) has been described with respect to the structure $\mathcal{A}_{\mathcal{SET}} = \langle \mathcal{S}, (\cdot)^{\mathcal{S}} \rangle$ [12]. The interpretation domain $\mathcal{S}$ is a subset of the set of (well-formed) ground terms built from symbols in $\mathcal{F}$, i.e., $T(\mathcal{F})$. Terms are partitioned into equivalence classes according to the set-theoretical properties of the

[1] We use the notation $\vec{X}$ to denote a sequence of variables.

symbol $\{\cdot\,|\,\cdot\}$; e.g., the term $\{a,b\}$ is considered equivalent to the term $\{b,a\}$. A complete discussion of these concepts is presented in Section 3.2, where a similar construction is developed for the new language CLP($\mathcal{SET},\mathcal{FD}$).

Example 3 *Let us assume that we represent an undirected graph as a set V of vertices and the set $E = \{\{\mu_1,\nu_1\},\{\mu_2,\nu_2\},\dots\}$, $\mu_i,\nu_i \in V$ of edges. The following CLP($\mathcal{SET}$) program computes the cliques of a given undirected graph $\langle V,E\rangle$:*

```
clique(V, E, Clique) :−
      Clique ⊆ V,
      (∀I ∈ Clique)(∀J ∈ Clique)({I, J} ∈ {{I} | E}).
```

The ability to use nested sets *and RUQs greatly facilitates the encoding of this problem. Observe that the set $\{I\}$ is required to account for cases where $I = J$.* □

The language CLP($\mathcal{SET}$) is endowed with a complete *constraint solver*, called $SAT_{\mathcal{SET}}$, for verifying the satisfiability of $\mathcal{SET}$-constraints. Given a constraint C, $SAT_{\mathcal{SET}}(C)$ transforms C either to false (if C is unsatisfiable) or to a finite collection $\{C_1,\dots,C_k\}$ of constraints in *solved form* (see also Section 4.3). A constraint in solved form is guaranteed to be satisfiable in the structure $\mathcal{A}_{\mathcal{SET}}$. Moreover, the disjunction of all the constraints in solved form generated by $SAT_{\mathcal{SET}}(C)$ is equisatisfiable to C w.r.t. $\mathcal{A}_{\mathcal{SET}}$. A detailed description of the constraint solver $SAT_{\mathcal{SET}}$ can be found in [12]; its implementation is included in the $\{log\}$-interpreter [25].

Example 4 *Let C be $\{1,2\,|\,X\} = \{1\,|\,Y\} \wedge 2 \notin X$. Then $SAT_{\mathcal{SET}}(C)$ returns the three constraints in solved form:*

- *C_1 is $Y = \{2\,|\,X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$;*
- *C_2 is $X = \{1\,|\,N\} \wedge Y = \{2\,|\,N\} \wedge \mathsf{set}(N) \wedge 2 \notin N$;*
- *C_3 is $Y = \{1,2\,|\,X\} \wedge 2 \notin X \wedge \mathsf{set}(X)$,*

where N is a new variable. Observe that: $\mathcal{A}_{\mathcal{SET}} \models \forall X \forall Y\,(C \leftrightarrow \exists N\,(C_1 \vee C_2 \vee C_3))$. □

2.3 CLP($\mathcal{FD}$)

The language CLP($\mathcal{FD}$) [7, 22, 28] is an instance of the CLP scheme, and it is particularly suited to encode combinatorial and optimization problems. The signature used by CLP($\mathcal{FD}$) contains [18]:

- a set $\mathcal{F}$ of constant and function symbols, including separate constants (e.g., $0,-1,1,-2,2,\dots$) to represent the elements of $\mathbb{Z}$, and function symbols to represent the standard arithmetic operations, such as $+,-,*,\mathsf{div},\mathsf{mod}$ etc.;
- the set of constraint predicate symbols $\Pi_C = \{\{\in m..n\},=,<,\leq\}$;
- a set Π_U of user-defined predicate symbols;
- a denumerable set $\mathcal{V}$ of variables.

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., global constraints such as `alldifferent`, 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 $\langle \Pi_C,\mathcal{F},\mathcal{V}\rangle$-literals.

The constraint $u \in m..n$ is concretely expressed as `u in m..n` in B-Prolog, in the library `clp/bounds` of SWI Prolog, and in the `clpfd` library of SICStus Prolog, as `fd_domain(u,m,n)` in GNU Prolog, and as `u #::m..n` in the library `ic` of ECLiPSe.

The interpretation structure $\mathcal{A}_{\mathcal{FD}}$ is based on

- an interpretation domain composed of the set $\mathbb{Z}$ of the integer numbers, and

- an interpretation function $(\cdot)^{\mathcal{FD}}$, which 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 $u \in m..n$ (a *domain declaration*) is: $u \in^{\mathcal{FD}} m..n$ holds iff $m \leq u \leq n$.

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 an equisatisfiable finite collection of $\mathcal{FD}$-constraints w.r.t. $\mathcal{FD}$. The resulting constraints cover the same solution space as the input constraint, but provide simplified constraints—in particular, intervals representing domains may have been narrowed.

The CLP($\mathcal{FD}$) solvers adopted by SICStus [6, 27], ECLiPSe [17], B-Prolog [33], SWI-Prolog [31] are incomplete solving procedures; e.g., given the goal [2]

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

these solvers do not determine the inconsistency. A complete solver can be obtained by adding 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 and `label([X,Y,Z])` in SWI-Prolog) then the constraint solver will successfully determine the unsatisfiability of the constraint. The predicate `labeling` is provided to force the assignment to the specified variables of values from their domains, leading to a chronological backtracking search of the space of solutions.

As summarized in Figure 1, the constraint solver of CLP($\mathcal{FD}$) with input a constraint C either returns a constraint C' implying C, or, via labeling, a canonical form solution. However, C' is possibly not in solved form and it is possibly unsatisfiable. This can be compared to the case of CLP($\mathcal{SET}$) (see Section 2.2), where the result obtained is either a disjunction of solved form satisfiable constraints C_i' or false (if C is unsatisfiable).

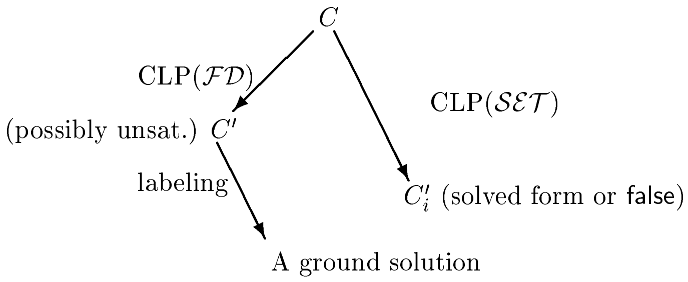

Figure 1: Incomplete and Complete Solvers

3 The Language CLP($\mathcal{SET}, \mathcal{FD}$)

In this section we present the syntax and the semantics of the Constraint Logic Programming language CLP($\mathcal{SET}, \mathcal{FD}$), which combines the two languages presented in the previous section.

3.1 Syntax

The signature $\Sigma = \langle \Pi_C \cup \Pi_U, \mathcal{F}, \mathcal{V} \rangle$ of the language CLP($\mathcal{SET}, \mathcal{FD}$) is defined as follows. Let $\mathcal{V}$ be a denumerable set of variables. The set of function symbols $\mathcal{F}$ allowed in CLP($\mathcal{SET}, \mathcal{FD}$) is

$$\mathcal{F} = \{\emptyset, \{\cdot \mid \cdot\}, \mathsf{int}\} \cup Z \cup F_Z \cup F_U$$

where

- $\emptyset$ and $\{\cdot \mid \cdot\}$ are the set constructors (as in Section 2.2);

[2]The symbol $\#\backslash =$ represents $\neq$, i.e., the negation of the predicate $=$.

- int is a binary set constructor used to build intervals;
- 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 $+, -, *, \mathsf{div}, \mathsf{mod}$, etc.;
- F_U is a (possibly empty) set of user-defined constant and function symbols, with the property that $F_U \cap (\{\emptyset, \{\cdot \mid \cdot\}, \mathsf{int}\} \cup Z \cup F_Z) = \emptyset$.

We assume the language to classify terms according to different sorts. We will make use of three sorts: Set (to represent sets), Int (to represent integer terms), and Ker (to represent user-defined terms). In particular,

- each term whose main functor is in F_U is of sort Ker (called *kernels* or *non-set terms* in [12]);
- (*integer constants*) each term in Z is of sort Int;
- (*set terms*) each term of the form $\{t \mid s\}$ is well-formed and of sort Set iff s is a well-formed term of sort Set;
- (*interval terms*) each term of the form $\mathsf{int}(a, b)$ is a well-formed term of sort Set iff a, b are well-formed terms of sort Int;
- (*compound integer terms*) each term of the form $f(t_1, \ldots, t_k)$, $f \in F_Z$, is well-formed and of sort Int iff $t_1, \ldots, t_k$ are well-formed terms of sort Int;
- *variables* are of sort Int $\cup$ Ker $\cup$ Set.

According to the given syntax, examples of well-formed terms are:

`int(1,2 * 10)`   `int(A,A + 10)`   `{X + 2 | int(1,10)}`

whereas `int(1,f(a))` and `a + 1` are not well-formed terms (`f`, `a` $\in F_U$). In the rest of the discussion we will concentrate exclusively on well-formed terms.

The set Π_C of constraint predicate symbols is

$$\Pi_C = \{=, \in, \cup_3, ||, \leq, \mathsf{set}, \mathsf{integer}\}$$

while Π_U (such that $\Pi_C \cap \Pi_U = \emptyset$) is the set of user-defined predicate symbols.

A *$\mathcal{SETFD}$-constraint* is a conjunction of $\langle \Pi_C, \mathcal{F}, \mathcal{V} \rangle$-literals. As illustrated in [12], all other useful set-theoretical predicates, e.g., $\subseteq, \cap_3$, can be defined as $\mathcal{SETFD}$-constraints, using $||$ and $\cup_3$—e.g.,

$$u \subseteq v \Leftrightarrow \cup_3(u, v, v)$$

Similarly, other interesting integer predicates (e.g., $<$, $\geq$, and $>$) can be defined as $\mathcal{SETFD}$-constraints using $\leq$ and $=$. We will use the notation $\neq$, $\notin$, $\not\cup_3$, $\ldots$, not_integer to denote the negated versions of the corresponding constraint predicates—e.g., $s \notin t$ represents the literal $\neg\,(s \in t)$.

With the notation $c[t_1, \ldots, t_n]$ we will indicate a generic constraint atom or literal which has the terms $t_1, \ldots, t_n$ as arguments.

Example 5 *The following CLP($\mathcal{SET}, \mathcal{FD}$)program is used to compute paths of length* `Max` *in an undirected weighted graph* `E` *having total cost within a given range* `CostMin` $\div$ `CostMax`*. The graph* `E` *is described as a set of edges where each edge is a set of the type* $\{$`Start`, `End`, `cost(Cost)`$\}$, `Start` *and* `End` *are the vertices of the edge, and* `Cost` *is its weight (an integer).*

```
find_path(E, Max, CostMin, CostMax, Path) :-
        group(Max, _ , Path, Cost),
        Path ⊆ E,
        Cost ∈ int(CostMin, CostMax).
group(0, _ , ∅, 0).
group(K, X, {{X, Y, cost(N)} | R}, N + M) :-
        K > 0,
        {X, Y, cost(N)} ∉ R,
        group(K − 1, Y, R, M).
```

The predicate `group/4` *is used to build an abstract path of the given length, along with the formula describing its cost. The* `find_path` *predicate operates as follows:*

1. *it builds the abstract path using the* `group` *predicate;*

2. *it ensures that the elements of the abstract path are edges of the graph* `E`*;*

3. *it ensures that the cost of the path is in the specified range.*

□

3.2 Semantics

The semantics of the language is an extension of the traditional minimal model semantics of (constraint) logic programming. The main extension deals with ensuring that constraint predicates are properly handled. In particular, we need to ensure that sets constructions are interpreted in such a way to respect the natural properties of sets (e.g., the ordering of elements in a set is irrelevant). This is accomplished by partitioning the set of terms into equivalence classes, where distinct terms representing the same set belong to the same equivalence class—and then using a representative of the equivalence class as result of the interpretation of any term belonging to such class.

Let us denote with $\mathbb{H}$ the Herbrand Universe built using symbols in $\mathcal{F} \setminus (\{\mathsf{int}\} \cup F_Z)$ and containing exclusively well-formed terms. Note that:

- we omit compound integer terms—as these are meant to be evaluated to the corresponding integer constants (elements of Z);
- we omit intervals, as these are meant to be unfolded into sets of integers.

Terms of $\mathbb{H}$ are partitioned into equivalence classes according to the set-theoretical properties of the symbol $\{\cdot \,|\, \cdot\}$ expressed by the two equational axioms:

$$\begin{array}{llcl} (Ab) & \{X \,|\, \{X \,|\, Z\}\} & = & \{X \,|\, Z\} \\ (C\ell) & \{X \,|\, \{Y \,|\, Z\}\} & = & \{Y \,|\, \{X \,|\, Z\}\} \end{array}$$

The axiom (Ab) states that duplicates in a set can be ignored (*Absorption property*). The axiom $(C\ell)$ states that the order of elements in a set is irrelevant (*Left Commutativity*). Let $\cong$ be the least congruence relation over $\mathbb{H}$ closed w.r.t. the equational axioms (Ab) and $(C\ell)$. This relation induces a partition of $\mathbb{H}$ into equivalence classes, and we will denote with $\mathbb{H}/\cong$ the collection of equivalence classes. For example, the terms $\{b, a\}$, $\{a, a, b\}$, and $\{a, b, b\}$ belong to the same equivalence class.

Let us assume that the symbols in F_U are totally ordered by a relation $\prec$. We extend $\prec$ to obtain a total order on $\mathcal{F} \setminus (\{\mathsf{int}\} \cup F_Z)$ in the following way:

1. $\emptyset \prec n$ for each $n \in Z$
2. $\cdots \prec -3 \prec -2 \prec -1 \prec 0 \prec 1 \prec 2 \prec 3 \prec \cdots$
3. if $n \in Z$ then $n \prec \{\cdot \,|\, \cdot\}$
4. if $n \in Z$ and $g \in F_U$ then $n \prec g$, and
5. if $f \in F_U$ then $f \prec \{\cdot \,|\, \cdot\}$.

Let us observe that $\emptyset$ and $\{\cdot \,|\, \cdot\}$ are respectively the minimum and maximum of the order relation $\prec$. With a slight abuse of notation, we can summarize the definition of $\prec$ as: $\emptyset \prec Z \prec F_U \prec \{\cdot \,|\, \cdot\}$. Condition (3) is superfluous if F_U is not empty. The order $\prec$ can be used to inductively define the total order $\lhd$ on the set $\mathbb{H}$, in the following way:

- If $f(s_1, \ldots, s_m)$ and $g(t_1, \ldots, t_n)$ are terms ($m \geq 0, n \geq 0$) and $f \prec g$ then $f(s_1, \ldots, s_m) \lhd g(t_1, \ldots, t_n)$;
- if $s_2 \lhd t_2$, or if $s_2 = t_2$ and $s_1 \lhd t_1$, then $\{s_1 \,|\, s_2\} \lhd \{t_1 \,|\, t_2\}$;
- If f is different from $\{\cdot \,|\, \cdot\}$, and
  - $s_1 \lhd t_1$, or
  - $s_1 = t_1$ and $s_2 \lhd t_2$, or

– ..., or
– $s_1 = t_1, s_2 = t_2, \ldots, s_{n-1} = t_{n-1}$ and $s_n \lhd t_n$

then $f(s_1, \ldots, s_n) \lhd f(t_1, \ldots, t_n)$.

Constant terms from Z denote integer numbers, and the order induced by $\lhd$ on them is the standard one. Terms with outermost function symbols in F_U are ordered lexicographically. Let us consider the ordering of terms denoting sets. If $m < n$ then $\{s_1, \ldots, s_m\} \lhd \{t_1, \ldots, t_n\}$. This implies that for terms representing sets containing the same elements, those without element repetitions are $\lhd$-smaller. Among them, the least one is the one having its elements in $\lhd$-decreasing order from left to right. For instance, assume that $F_U = \{a, b, c\}$ and $a \prec b \prec c$. Then $\{c, b, a\} \lhd \{c, b, a, a\}$ (since $\emptyset \lhd \{a\}$) and $\{c, b, a\} \lhd \{a, b, c\}$ (since $a \lhd c$).

For each equivalence class in $\mathbb{H}/\cong$, we can uniquely identify the $\lhd$-minimum term in the class. Given a term $t \in \mathbb{H}$, with $t^{\mu}_{\lhd}$ we denote the $\lhd$-minimum term in the equivalence class of t. Observe that if the term t does not contain any set term, then $t^{\mu}_{\lhd} = t$.

The semantics of CLP($\mathcal{SET}, \mathcal{FD}$) is described with respect to the structure $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} = \langle A^{\mathcal{SET}}_{\mathcal{FD}}, I \rangle$ where:

$$A^{\mathcal{SET}}_{\mathcal{FD}} = \{ t \in \mathbb{H} \,:\, t = t^{\mu}_{\lhd} \}$$

Let us observe that $A^{\mathcal{SET}}_{\mathcal{FD}} \subseteq \mathbb{H}$, and thus the semantic notion of domain element can be concretely rendered by the syntactic notion of term. Moreover, since $0, -1, 1, 2, -2, \ldots$ are constant terms in $A^{\mathcal{SET}}_{\mathcal{FD}}$, $A^{\mathcal{SET}}_{\mathcal{FD}}$ contains a "copy" of the set $\mathbb{Z}$ of the integer numbers. Each function symbol $f \in F_Z$ corresponds to a (standard) operation on integer numbers. Let us denote with $f^{\mathbb{Z}}$ this operation.[3] For example, $+^{\mathbb{Z}}$ is the function that performs addition between two integer numbers.

We define a partition of $A^{\mathcal{SET}}_{\mathcal{FD}}$ in three sets:

$$\begin{array}{lcl} \texttt{Int} & = & \{0, -1, 1, -2, 2, \ldots\} = Z \\ \texttt{Set} & = & \{\emptyset\} \cup \{ \{t \,|\, s\}^{\mu}_{\lhd} \,:\, t \in \mathbb{H}, s \in \texttt{Set} \} \\ \texttt{Ker} & = & A^{\mathcal{SET}}_{\mathcal{FD}} \setminus (\texttt{Int} \cup \texttt{Set}) \end{array}$$

We can now proceed in the definition of the *interpretation function* I:

- $I(\emptyset) = \emptyset$;
- For each $n \in Z$, $I(n) = n$;
- $I(\mathsf{int}(a, b)) = \begin{cases} I(\{I(a), I(a+1), I(a+2), \ldots, I(b)\}) & \text{if } I(a) \in \texttt{Int}, I(b) \in \texttt{Int}, \\ & \text{and } I(a) \lhd I(b) \text{ or } I(a) = I(b) \\ \emptyset & \text{otherwise;} \end{cases}$
- For $f \in F_Z$, if $I(t_1), \ldots, I(t_n) \in \texttt{Int}$ then $I(f(t_1, \ldots, t_n)) = f^{\mathbb{Z}}(I(t_1), \ldots, I(t_n))$ (namely, the integer function is applied to integers);
- For $f \in F_U$, $I(f(t_1, \ldots, t_n)) = f(I(t_1), \ldots, I(t_n))$;
- Let us consider n terms $t_1, \ldots, t_n$. Then, $I(\{t_1, \ldots, t_n\}) = \{I(t_1), \ldots, I(t_n)\}^{\mu}_{\lhd}$.

Intuitively, the interpretation of a term is its $\lhd$-representative in $\mathbb{H}/\cong$ after all the occurrences of int have been removed and all the arithmetic functions have been evaluated.

Integer terms are interpreted on the subset of the domain of $\mathcal{SETFD}$ containing integer constants (i.e., on Int). Arithmetic function symbols are interpreted according to their traditional mathematical meaning.

The interpretation I of the predicate symbols in Π_C is given below. Given a n-ary predicate $p \in \Pi_C$ we will denote with p^I its interpretation, where $p^I : {A^{\mathcal{SET}}_{\mathcal{FD}}}^n \to \{\mathsf{true}, \mathsf{false}\}$. Let r, s, t denote well-formed terms in $A^{\mathcal{SET}}_{\mathcal{FD}}$. Then

- $I(s = t)$ is true iff s and t represent the same element in $A^{\mathcal{SET}}_{\mathcal{FD}}$—i.e., $=^I$ is the syntactic equality relation over $A^{\mathcal{SET}}_{\mathcal{FD}}$;
- $I(s \in t)$ is true iff $t \in \texttt{Set}$, t is of the form $\{s_1, \ldots, s_i, \ldots, s_n\}$ and $s_i = s$ for some $i \in \{1, \ldots, n\}$;
- $I(\cup_3(r, s, t))$ is true iff $r, s, t \in \texttt{Set}$, $r = \{r_1, \ldots, r_m\}$, $s = \{s_1, \ldots, s_n\}$, and $t = I(\{r_1, \ldots, r_m, s_1, \ldots, s_n\})$;
- $I(s \,||\, t)$ is true iff $r, s \in \texttt{Set}$, $r = \{r_1, \ldots, r_m\}$, $s = \{s_1, \ldots, s_n\}$, and for all r_i and s_j it holds that $r_i \neq s_j$;

[3] With a slight abuse of notation, we assume that the result of each $f^{\mathbb{Z}}$ is actually a constant of $\mathbb{H}$ representing the integer result.

- $I(s \leq t)$ is true iff $s, t \in \texttt{Int}$ and $s \lhd t$ or $s = t$;
- $\mathsf{integer}(s)$ holds iff $s \in \texttt{Int}$;
- $\mathsf{set}(s)$ holds iff $s \in \texttt{Set}$.

Observe that the language CLP($\mathcal{SET}, \mathcal{FD}$) is an instance of CLP($\mathcal{SET}$), where the underlying signature contains an infinite number of constants (Z). The additional functional symbol int is effectively just a syntactic sugar for sets constructed using elements of Z. The novelty is represented by the availability of integer constraints and integer operations.

The semantics of clauses and programs follows from the semantics just presented for terms and predicates, as usual in CLP [12, 18].

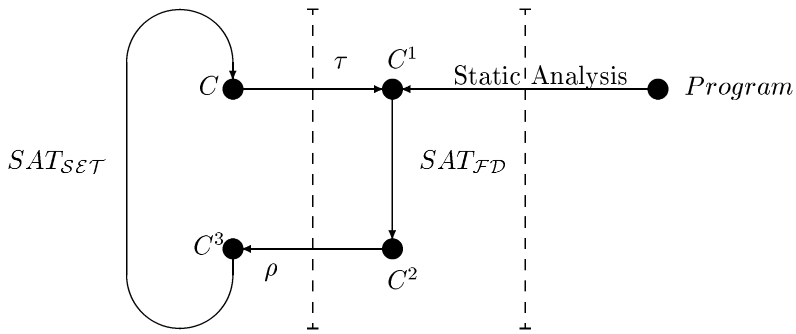

Figure 2: Schema of the $SAT_{\mathcal{SET}+\mathcal{FD}}$ Constraint Solving

4 Constraint Solving

In this section we describe how the process of handling constraints of CLP($\mathcal{SET}, \mathcal{FD}$) is organized. The solution we propose is to exploit the CLP($\mathcal{FD}$) constraint solver within the $SAT_{\mathcal{SET}}$ constraint solver—the resulting combined solver is called $SAT_{\mathcal{SET}+\mathcal{FD}}$. Figure 2 provides a general overview of the structure of the constraint solving process:

1. A $\mathcal{SETFD}$-constraint C (top-left node in Fig. 2) is partially rewritten by a function τ into a $\mathcal{FD}$-constraint C^1. The function τ extracts from C those constraints that can be effectively expressed as $\mathcal{FD}$ constraints.

2. The CLP($\mathcal{FD}$) solver reduces C^1 to a constraint C^2, and eventually returns it to $SAT_{\mathcal{SET}}$ using a conversion function ρ, which produces the CLP($\mathcal{SET}$) constraint C^3. The job of the CLP($\mathcal{FD}$) solver could be simplified by the availability of information statically gathered from the program *Program*—as described in Section 5.

3. The $SAT_{\mathcal{SET}}$ procedure processes this resulting constraint C^3 and repeats the process.

The process continues until no further simplifications of the constraints are possible.

In this section, we define and analyze the functions τ and ρ, the various procedures of $SAT_{\mathcal{SET}}$ used in the constraint solver, and the required modifications w.r.t. the original procedures presented in [12], to allow communication between the two solvers. In particular, in Subsection 4.1 we introduce the partitioning of the constraints between the two solvers. Subsection 4.2 describes the translation functions τ and ρ, while Subsection 4.3 presents the new notion of *solved form* for the constraints. Subsections 4.4 and 4.5 describe the constraint solving procedures.

4.1 Three Families of Constraints

We identify three different types of constraints that are dealt with by $SAT_{\mathcal{SET}+\mathcal{FD}}$. Precisely, each $\mathcal{SETFD}$-constraint C is split into three parts: $C = C^F \wedge C^C \wedge C^S$, where:

C^F is handled only by a $\mathcal{FD}$ solver: C^F contains

- the primitive constraints of the type $s \in \mathsf{int}(t_1, t_2)$ or $s \leq t$,
- constraints of the type $s \in \{t_1, \ldots, t_n\}$, where $t_1, \ldots, t_n$ are ground integer terms,
- constraints of the type $s = t$ or $s \neq t$, where either s or t are compound integer terms.

C^C is handled by both solvers: C^C contains the primitive constraints of the type $s = t$ or $s \neq t$, where s, t are either variables or integer constants.

C^S is handled only by the $\mathcal{SET}$ solver: C^S contains all remaining constraints (e.g., symbolic user-defined constraints, non-interval set constraints).

Example 6 *Consider the constraint:*

$$C \equiv \{1, \{\emptyset, 1\}\} = \{\{Y, X\}, X\} \wedge X \leq 5 \wedge Y \neq X + 2 \wedge X \neq 3 \wedge X \neq Y \wedge X \in \mathsf{int}(2, 6)$$

then

- C^S *is* $\{1, \{\emptyset, 1\}\} = \{\{Y, X\}, X\}$*;*
- C^F *is* $X \leq 5 \wedge Y \neq X + 2 \wedge X \in \mathsf{int}(2, 6)$*;*
- C^C *is* $X \neq 3 \wedge X \neq Y$*.*

□

The C^C constraint effectively represents a communication conduit between the two constraint solvers. The intuition is that the constraints present in C^C will be continuously exchanged between $SAT_{\mathcal{SET}}$ and $SAT_{\mathcal{FD}}$, and possibly simplified by both solvers.

Remark 7 *Let us observe that a direct treatment of C^F constraints in CLP($\mathcal{SET}$) can be realized assuming that (non-negative) integer numbers are encoded as sets. For instance, if integers are encoded* à la *Von Neumann:*

$$\begin{cases} \underline{0} & = & \emptyset, \\ \underline{i+1} & = & \{\underline{i} \,|\, \underline{i}\} \end{cases}$$

then the constraint $t \leq s$ can be simply expressed as $t \in \{s \,|\, s\}$. In this case the introduction of the predicate symbol $\leq$ in CLP($\mathcal{SET}, \mathcal{FD}$) could be seen as a syntactic sugar. This approach is, clearly, very inefficient.

4.2 The Translation Functions τ and ρ

The function τ is in charge of mapping constraints of CLP($\mathcal{SET}, \mathcal{FD}$) to constraints that are valid inputs for the CLP($\mathcal{FD}$) solver. In our discussion, we assume a generic CLP($\mathcal{FD}$) solver, which accepts the following syntax for its primitive constraints:

- Domain declarations are expressed as $t \in (a_1 \mathinner{..} b_1) \backslash\!/ (a_2 \mathinner{..} b_2) \backslash\!/ \cdots \backslash\!/ (a_n \mathinner{..} b_n)$, where a_i, b_i are integer constants and $a_i \preceq b_i$. If $a_i = b_i$ then the interval $a_i \mathinner{..} b_i$ is simply denoted by the singleton set $\{a_i\}$.
- Equality and inequality constraints are expressed as $s = t$ and $s \neq t$.
- Comparison constraints are expressed as $s \leq t$.

4.2.1 The Function τ

Let c be a primitive $\mathcal{SETFD}$-constraint. The translation function τ can be defined as follows:

$$\tau(c) = \begin{cases} s \in t_1..t_2 & \textbf{if } c \equiv s \in \mathsf{int}(t_1, t_2), \text{ where } t_1, t_2 \text{ are ground integer terms} \\ & \text{ and } s \text{ is a variable or a term of sort } \mathsf{Int}; \\ s \in \{t_1\}\backslash/ \cdots \backslash/\{t_n\} & \textbf{if } c \equiv s \in \{t_1, \ldots, t_n\}, \text{ where } t_1, \ldots, t_n \text{ are ground integer terms} \\ & \text{ and } s \text{ is a variable or a term of sort } \mathsf{Int}; \\ c & \textbf{if } c \textit{ is } s = t \textit{ or } s \neq t \textit{ or } s \leq t, \text{ where } s \text{ and } t \text{ are variables} \\ & \text{ or terms of sort } \mathsf{Int}; \\ \mathsf{true} & \textit{otherwise.} \end{cases}$$

Note that the function τ simply returns true every time a constraint in C^S is mapped to the CLP($\mathcal{FD}$) solver. E.g.,

$$\tau(\ f(t_1) = f(t_2)\ ) = \mathsf{true} \qquad\qquad \tau(\ \cup_3(t_1, t_2, t_3)\ ) = \mathsf{true}$$

where $f \in F_U$ and t_1, t_2, t_3 are terms.

In concrete syntax, the $\mathcal{SETFD}$-constraint $s \in \mathsf{int}(a, b)$ will correspond to the SICStus $\mathcal{FD}$-constraint `s in a..b`. If s is a compound term, one should write `S #= s, S in a..b`. Moreover, $s \neq t$, $s \leq t$, and $s = t$ will correspond to the constraints `s #\= t`, `s #=< t`, and `s #= t`, respectively.

Proposition 8 *Let C be a $\mathcal{SETFD}$-constraint consisting of a conjunction of literals of the forms*

$$\ell \in \mathsf{int}(t_1, t_2), \ell \in \{t_1, \ldots, t_n\}, \ell = r, \ell \neq r, \ell \leq r$$

where t_i are ground integer terms and ℓ, r are either variables or terms of sort Int. *Then*

$$\mathcal{FD} \models \vec{\exists}\tau(C) \textit{ iff } CLP(\mathcal{SET}, \mathcal{FD}) \models \vec{\exists}C$$

Proof: Since $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}}$ contains a sub-model isomorphic to $\mathcal{FD}$, the ($\rightarrow$) direction follows trivially. For the ($\leftarrow$) direction, assume that a valuation σ of the variables in C in the domain D is such that $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \sigma(C)$. We show how to construct a valuation σ' of the variables in C to $\mathbb{Z}$ such that $\mathcal{FD} \models \sigma'(\tau(C))$. Let M be the maximum integer occurring in the various literals present in $\sigma(C)$.

- If $\sigma(X) \in$ `Int` then let $\sigma'(X) = \sigma(X)$;
- Otherwise, we are guaranteed that X does not occur in a constraint of the form $X \in \mathsf{int}(t_1, t_2)$ or $X \in \{t_1, \ldots, t_n\}$. Assign $\sigma'(X) = M + 1$ (and $\sigma'(Y) = M + 1$, for all other variables Y such that $\sigma(X) = \sigma(Y)$).

The construction continues by updating M to $M + 1$ and repeating the above step for all the variables. $\square$

Hence, we can use the CLP($\mathcal{FD}$) constraint solver to deal with primitive constraints involving intervals and constraints of the kind $\ell = r$, $\ell \neq r$—where only arithmetic functions are in ℓ or in r—and disequations of the form $\ell \leq r$.

4.2.2 The Function ρ

The function ρ returns the equality and inequality $\mathcal{FD}$-constraints back to the $SAT_{\mathcal{SET}}$ solver. Let c be a primitive $\mathcal{FD}$-constraint:

$$\rho(c) = \begin{cases} X = t & \textbf{if } c \equiv X = t \text{ and } t \text{ is an integer constant or a variable;} \\ X \neq t & \textbf{if } c \equiv X \neq t \text{ and } t \text{ is an integer constant or a variable;} \\ \mathsf{true} & \textit{otherwise.} \end{cases}$$

Basically, ρ is the inverse of the function τ, restricted to the simplified equality and inequality constraints (actually, ready to be converted in *solved form*—see Definition 9). Membership constraints (such as $X \in 1..3$) are not returned to the $SAT_{\mathcal{SET}}$ solver, since they would be immediately solved by enumeration. We prefer to leave these constraints inside the $\mathcal{FD}$ solver, delaying the explicit enumeration of solutions (e.g., using the predicate `labeling` to generate a chronological backtracking search) until not strictly necessary to guarantee completeness of the constraint solving procedure. We prefer to wait for this enumeration to be explicitly requested by the programmer (e.g., using the predicate `labeling` to generate a chronological backtracking search). The functions τ and ρ are extended in the natural way to deal with conjunctions of primitive constraints.

We will also refer to the inverse function τ^{-1}, where, in particular,

$$\begin{array}{rcl} \tau^{-1}(s \in t_1..t_2) & = & s \in \mathsf{int}(t_1, t_2) \\ \tau^{-1}(s \in (a_1..b_1)\bigvee(a_2..b_2)\bigvee\cdots\bigvee(a_n..b_n)) & = & s \in \{a_1, a_1+1, \ldots, b_1, a_2, a_2+1, \ldots, b_2, \ldots, a_n, a_n+1, \ldots, b_n\} \end{array}$$

For the sake of simplicity in the presentation, we assume that terms and variables are shared by CLP($\mathcal{SET}, \mathcal{FD}$) and CLP($\mathcal{FD}$), i.e., the function τ passes them between the two solvers without any modifications. We skipped the definition of an additional mapping between individual variables, even though, from a theoretical point of view, every constraint solver is treated as a black box, and its terms are inaccessible from the other solvers. Nevertheless, this abuse of notation is not exploited to infer any kind of implicit propagation of variables and terms among different solvers. For example, every time a variable is instantiated in CLP($\mathcal{FD}$), this operation reflects the update in CLP($\mathcal{SET}, \mathcal{FD}$) only via the function ρ, that adds the equality constraint to C^S. This approach to integration of solvers is rather common, see e.g., [16]. Note that, on the other hand, in our implementation, we are able to access both solvers information and we actually share variables and terms.

4.3 Solved Form

The constraint solving process, described in the next section, returns as result a pair $\langle S, D\rangle$, where S is a $\mathcal{SET}$-constraint in *solved form*, and D is a $\mathcal{FD}$-constraint as returned by $SAT_{\mathcal{FD}}$. The solved form for $\mathcal{SETFD}$-constraints is derived from the notion of solved form for $\mathcal{SET}$-constraints, introduced in [12].

Definition 9 Let C be a $\mathcal{SETFD}$-constraint. A literal c of C is in *solved form* if it satisfies one of the following conditions:

(i) c is of the form $X = t$, where X is a variable, t is a CLP($\mathcal{SET}, \mathcal{FD}$) term, and neither t nor $C \setminus \{c\}$ contain X;

(ii) c is of the form $X \neq t$, where X is a variable, t is a CLP($\mathcal{SET}, \mathcal{FD}$) term, and X does not occur in t;

(iii) c is of the form $t \notin X$, where X is a variable, t is a CLP($\mathcal{SET}, \mathcal{FD}$) term, and X does not occur in t;

(iv) c is of the form $\cup_3(X_1, X_2, X_3)$, where X_1, X_2, X_3 are variables, $X_1 \not\equiv X_2$, and for $i = 1, 2, 3$ there are no disequations of the form $X_i \neq t$ or $t \neq X_i$ in C;

(v) c is of the form $X_1 \parallel X_2$, where X_1, X_2 are distinct variables;

(vi) for each variable X, at most one among: $\mathsf{set}(X)$, $\mathsf{integer}(X)$, $\mathsf{not\_integer}(X)$ is in C.

A constraint C is in solved form if it is empty or if all its components are simultaneously in solved form.

Remark 10 *Following the approach used in CLP($\mathcal{SET}$), constraints of the form* not_set *are not allowed. Their inclusion however, would require very few changes. As far as the above solved form definition is concerned, we would need to accept the conjunction* $\mathsf{not\_set}(X_i) \wedge \mathsf{not\_integer}(X_i)$ *for a variable* X_i*, which is a hint that* X_i *represents a term of sort* Ker*. Few changes are needed in the proof of the following lemma, where a further case* $X_i \mapsto f(n_i)$ *has to be considered in point (1).*

Lemma 11 *A $\mathcal{SETFD}$-constraint C in solved form is satisfiable in the structure $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}}$.*

Proof: In the proof we use the auxiliary function *find*:

$$\mathit{find}(x,t) \quad = \quad \begin{cases} \emptyset & \text{if } t = \emptyset,\ x \neq \emptyset; \\ \{0\} & \text{if } t = x; \\ \{1+n : n \in \mathit{find}(x,y)\} & \text{if } t = \{y \,|\, \emptyset\}; \\ \{1+n : n \in \mathit{find}(x,y)\} \cup \mathit{find}(x,s) & \text{if } t = \{y \,|\, s\},\ s \neq \emptyset. \end{cases}$$

which returns the set of "depths" at which a given element x occurs in the set t. Also, We will refer to

$$\overbrace{\{\cdots\{\emptyset\}\cdots\}}^{n_i}$$

as $\{\emptyset\}^{n_i}$.

The proof relies on the construction of a mapping for the variables of C into $\mathcal{A}_{\mathcal{FD}}^{\mathcal{SET}}$. The construction is divided into two parts. In the first part, we will not consider the constraints based on the $=$ predicate (let us denote with $C_=$ such set of constraints). A solution for the other constraints is computed by looking for valuations of the form

$$X_i \mapsto \underbrace{\{\cdots\{\emptyset\}\cdots\}}_{n_i} \quad \text{or} \quad X_i \mapsto n_i \tag{1}$$

fulfilling all the $\neq$, $||$, $\notin$, and $\cup_3$ constraints, as well as the type constraints set, $\mathsf{integer}$ and $\mathsf{not\_integer}$. In particular, we determine the valuation for each X_i according to the presence of the type constraint associated to X_i in C. The variables appearing in $\cup_3$ are mapped into $\emptyset$ ($n_i = 0$) and the numbers n_i for the other variables are computed choosing one possible solution of a system of integer equations and disequations, that trivially admits solutions. Such system is obtained by analyzing the "depth" of the occurrences of the variables in the set terms; in this case, the $||$- and $\neq$-constraints are treated in the same way. For $\mathsf{integer}$ variables, additional disequations are added, depending on the integer terms appearing in the constraints. Then, all the variables occurring only in the right-hand side of equations of $C_=$ are bound to $\emptyset$, while the variables present in the left-hand side are bound according to the uniquely induced valuation.

In detail, let $X_1, \ldots, X_m$ be all the variables occurring in C, except for those occurring in the left-hand side of equalities; let $X_1, \ldots, X_h$, $h \leq m$ be those variables occurring in $\cup_3$-atoms. In addition, let $n_1, \ldots, n_m$ be auxiliary variables ranging over $\mathbb{N}$. Let us construct the system *Syst* as follows:

- For all $i \leq h$, add the equation $n_i = 0$;
- For all $h < i \leq m$, add the following disequations:
  - if there are not type constraints for X_i or if there is the constraint $\mathsf{set}(X_i)$ or the constraint $\mathsf{not\_integer}(X_i)$, then

$$\begin{array}{ll} n_i \neq n_j + c & \forall X_i \neq t \text{ in } C \text{ and } c \in \mathit{find}(X_j, t) \\ n_i \neq c & \forall X_i \neq t \text{ in } C \text{ and } t \equiv \{\emptyset\}^c \\ n_i \neq n_j + c + 1 & \forall t \notin X_i \text{ in } C \text{ and } c \in \mathit{find}(X_j, t) \\ n_i \neq c + 1 & \forall t \notin X_i \text{ in } C \text{ and } t \equiv \{\emptyset\}^c \\ n_i \neq n_j & \forall X_i \,||\, X_j \text{ in } C \end{array}$$

  - if there is the constraint $\mathsf{integer}(X_i)$ then

$$n_i \neq t \quad \forall X_i \neq t \text{ in } C \text{ and } t \in Z;$$

- For all $h < i \leq m$ and $h < i \leq m$, such that $\mathsf{integer}(X_i)$ and $\mathsf{integer}(X_j)$, add the disequation:

$$n_i \neq n_j$$

If $m = h$, then $n_i = 0$ for all $i = 1, \ldots, m$ is the unique solution of *Syst*. Otherwise, it is easy to observe that it admits infinitely many solutions. In particular,

- Let $\{n_1 = 0, \ldots, n_h = 0, n_{h+1} = \bar{n}_{h+1}, \ldots, n_m = \bar{n}_m\}$ be one arbitrarily chosen solution of *Syst*.
- Let θ be the valuation such that:

$$\theta(X_i) = \begin{cases} n_i & \text{for all } h < i \leq m \text{ s.t. } \mathsf{integer}(X_i) \text{ is in } C \\ \{\emptyset\}^{n_i} & \text{otherwise} \end{cases}$$

- Let $Y_1, \ldots, Y_k$ be all the variables of C which appear only on the l.h.s. of equalities of the form $Y_i = t_i$.

- Let σ be the valuation such that $\sigma(Y_i) = \theta(t_i)$.

We can prove that $\mathcal{A}_{\mathcal{SET}} \models C[\theta\sigma]$, by case analysis on the structure of the literals in C:

1. $Y_i = t_i$: It is satisfied, since $\sigma(Y_i)$ has been defined as a ground term and equal to $\theta(t_i)$.

2. $X_i \neq t$: If there is not a constraint $\mathsf{integer}(X_i)$ and t is a ground term, then we have two cases:

   (a) if t is not of the form $\{\emptyset\}^c$, then it is obvious that $\theta(X_i) \neq t$;

   (b) if t is of the form $\{\emptyset\}^c$, for some c, then we have $n_i \neq c$, by construction, and hence $\theta(X_i) \neq t$.

   If there is not a constraint $\mathsf{integer}(X_i)$ and t is not ground, then if $\theta(X_i) = \theta(t)$, then there exists a variable X_j in t such that $\bar{n}_i = \bar{n}_j + c$ for some $c \in \mathit{find}(X_j, t)$; this cannot be the case since we started from a solution of $Syst$.

   If $\mathsf{integer}(X_i)$ and t is a ground term (i.e. t is integer), then $\theta(X_i) \neq t$ by construction.

   If $\mathsf{integer}(X_i)$ and t is not ground, t is a variable X_j for some j, by definition of solved form. By construction, $n_i \neq n_j$ and $\theta(X_i) \neq \theta(t)$.

3. $t \notin X_i$: Similar to the case above. Note that if t is an integer term, its valuation is a number, while $\theta(X_i)$ is a set, thus $\theta(t) \notin \theta(X_i)$.

4. $\cup_3(X_i, X_j, X_k)$: This means that $\bar{n}_i = \bar{n}_j = \bar{n}_k = 0$ and $\theta(X_i) = \theta(X_j) = \theta(X_k) = \emptyset$.

5. $X_i \,||\, X_j$: If $i, j \leq h$, then $\theta(X_i) = \theta(X_j) = \emptyset$.
   If $i > h$ (the same if $j > h$), then $\bar{n}_i \neq \bar{n}_j$, and hence $\theta(X_i) = \{\emptyset\}^{\bar{n}_i}$ is disjoint from $\theta(X_j) = \{\emptyset\}^{\bar{n}_j}$.

6. $\mathsf{integer}(X_i), \mathsf{set}(X_i), \mathsf{not\_integer}(X_i)$: Type constraints are trivially satisfied since mappings are built according to their presence.

□

Remark 12 *The constraint solver we describe in the next section is capable of dealing with intervals whose end points can be either ground or non-ground integer terms for all the possible constraints, except for* $||$ *and* $\cup_3$ *which, in contrast, require both end points to be ground. If this is not the case, constraints of the form:*

- $\mathsf{int}(s,t) \,||\, Y$*, where* s *or* t *(or both) are non-ground, and*
- $\cup_3(u, v, z)$*, where any of* u, v, z *are of the form* $\mathsf{int}(s,t)$*, with* s *or* t *(or both) non-ground integer terms, and the remaining of* u*,* v*, and* z *are variables,*

are left in the returned constraint C*. In this case, we cannot guarantee the satisfiability. For example, a possible constraint returned by the CLP(*$\mathcal{SET}, \mathcal{FD}$*) solver is*

$$1 \notin X \wedge 1 \notin Y \wedge \cup_3(X, Y, \mathsf{int}(Z, 1)) \wedge \mathsf{integer}(Z) \wedge \mathsf{set}(X) \wedge \mathsf{set}(Y)$$

which is clearly unsatisfiable.

Our solver is therefore not complete. Observe that this aspect is not surprising, as CLP($\mathcal{SET}, \mathcal{FD}$*) is based on the use of incomplete CLP(*$\mathcal{FD}$*) solvers to handle integer constraints. In some cases, further processing could be done on these constraints. For instance, from* $\cup_3(\mathsf{int}(2, A), \mathsf{int}(B, 5), \mathsf{int}(C, 8))$ *we can infer* $(A = 8 \wedge C = 2) \vee (A = 8 \wedge C = B \wedge C \leq 2)$*. Other cases, however, are more difficult to handle. For instance, in* $\cup_3(\mathsf{int}(A, B), C, \mathsf{int}(D, E))$ *we can infer* $D \leq A \wedge B \leq E$*. However, the union constraint cannot be removed or simplified.*

Procedure $SAT_{\mathcal{SET}+\mathcal{FD}}(C)$:
C:=$\mathsf{set\_int\_infer}(C)$;
$\langle S, D\rangle$:=$\langle C, \mathsf{true}\rangle$;
repeat
$\langle S', D'\rangle := \langle S, D\rangle$;
$\langle S, D\rangle := \mathsf{STEP}(\langle S, D\rangle)$;
until $\langle S, D\rangle = \langle S', D'\rangle$;
return $\langle S, D\rangle$

Figure 3: The $SAT_{\mathcal{SET}+\mathcal{FD}}$ Constraint Solver

4.4 The Global Constraint Solver

The overall structure of the constraint solver $SAT_{\mathcal{SET}+\mathcal{FD}}$ is illustrated in Figure 3. Constraints dealt with by $SAT_{\mathcal{SET}+\mathcal{FD}}$ are organized as pairs $\langle S, D\rangle$ (intuitively, S stands for Sets and D for finite Domains), where

- $S = C^S \wedge C^C$, and
- $D = \tau(C^F \wedge C^C)$.

D represents the constraint store where the constraints that are processed by the $SAT_{\mathcal{FD}}$ solver are accumulated. The procedure $SAT_{\mathcal{SET}}$ is modified to handle pairs of constraints of this form, and to properly distribute the different components to the proper solvers—i.e., the part S will be handled by the modified $SAT_{\mathcal{SET}}$ solver while the part D will be handled by $SAT_{\mathcal{FD}}$.

$SAT_{\mathcal{SET}+\mathcal{FD}}$ makes use of two procedures to accomplish its task: set_int_infer and STEP. The procedure set_int_infer is used to add set and integer constraints for those variables that are required to be of such sorts. The STEP procedure is the core part of $SAT_{\mathcal{SET}+\mathcal{FD}}$: it calls the rewriting procedures on the current constraint, represented as a pair $\langle S, D\rangle$. At the end of the execution of the STEP procedure, non-solved primitive constraints may still occur in $\langle S, D\rangle$; therefore, the execution of STEP has to be iterated until a fixpoint is reached—i.e., the constraint cannot be simplified any further. STEP and set_int_infer will be described in detail in the next subsection.

As mentioned in Section 2.2, if C is satisfiable, then $SAT_{\mathcal{SET}}(C)$ non-deterministically returns satisfiable constraints in solved form; if C is not satisfiable, then the solver simply returns false. This is not the case for the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver. Let $\langle S', D'\rangle$ be one of the pairs returned by the procedure $SAT_{\mathcal{SET}+\mathcal{FD}}$. $S' \wedge \tau^{-1}(D')$ is no longer guaranteed to be satisfiable in $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}}$. For example, $SAT_{\mathcal{SET}+\mathcal{FD}}$ on the unsatisfiable constraint:

$$X \notin S \ \wedge\ X \neq Y \ \wedge\ X \neq Z \ \wedge\ Y \neq Z \ \wedge\ X \in \mathsf{int}(1,2) \ \wedge\ Y \in \mathsf{int}(1,2) \ \wedge\ Z \in \mathsf{int}(1,2)$$

returns the pair

$$\left\langle \begin{array}{lcl} X \notin S \ \wedge\ X \neq Y \ \wedge\ X \neq Z \ \wedge\ Y \neq Z & , & X \neq Y \ \wedge\ X \neq Z \ \wedge\ Y \neq Z \ \wedge\ X \in 1..2 \ \wedge \\ & & X \in 1..2 \ \wedge\ Y \in 1..2 \ \wedge\ Z \in 1..2 \end{array} \right\rangle$$

instead of returning false. On the other hand, observe that $SAT_{\mathcal{SET}+\mathcal{FD}}$ still preserves the set of solutions of the original constraint, as proved in Proposition 8.

It is possible to obtain a complete $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver, when the constraints in D, returned by the solver, are guaranteed to be satisfiable. The completeness can be achieved by introducing an explicit enumeration of the solutions in $SAT_{\mathcal{FD}}$—e.g., by using backtracking to force exploration of the solution space (e.g., using the labeling predicate).

Specifically, let C be a $\mathcal{SETFD}$-constraint as in Proposition 8 and let $\mathcal{V}_{\in}$ be the set of variables occurring in the $\in$-constraints. If $\mathcal{V}_{\in} = \mathit{vars}(C)$, then it is possible to explore every ground solution (or detect unsatisfiability) for C. In this case, consistent domain values are assigned to variables in

$vars(C)$ by the CLP($\mathcal{FD}$) solver—by determining the ground solution (or unsatisfiability) of the constraint $\tau(C) \wedge$ `labeling`$([vars(C)])$—and returned to the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver.

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—as it will effectively lead to a translation of each interval into an extensional set. On the other hand, in our approach, we try to achieve a behavior for the global solver 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 approach we propose consists of a global constraint solving procedure, which uses $SAT_{\mathcal{SET}+\mathcal{FD}}$ and which exploits labeling techniques to enumerate all the solutions for the given constraint. In practice, however, the global constraint solving procedure will be called only when strictly necessary (e.g., at the end of the computation involved by the solution of a goal G with respect to a CLP($\mathcal{SET},\mathcal{FD}$) program P), in order to not compromise execution efficiency. The global constraint solving procedure is shown in Figure 4. Steps (2) and (3) are non-deterministic. In particular, if $SAT_{\mathcal{SET}+\mathcal{FD}}(S' \wedge \rho(R_i))$ returns false, then a different R_i will be considered. If all the R_i's have been attempted, then the global constraint solving procedure returns false result.

In Section 4.6 (Theorem 17), we will show that, if $\mathcal{V}_\in = vars(C)$, the global solving procedure returns a pair $\langle S', D'\rangle$, where S' is in solved form (cf. Remark 12), and thus the procedure is complete. Note that if $\mathcal{V}_\in \neq vars(C)$, some unbound variable domains may be left in the problem, and thus the completeness requirement can be reached only by means of exploration of a (possibly infinite) number of alternatives. Static analysis and in particular domain analysis proposed in Section 5 are aimed at obtaining membership constraints that are not explicitly defined by the user. This additional information can satisfy $\mathcal{V}_\in = vars(C)$ requirement.

Procedure Global_$SAT_{\mathcal{SET}+\mathcal{FD}}$(C)
    let $\langle S', D'\rangle$ be one of the solutions returned by $SAT_{\mathcal{SET}+\mathcal{FD}}(C)$;
    if no solution exists then **return** false;
    repeat until (there are no constraints of the form $X \in t_1..t_2$ in D')
(1)   extract any $X \in t_1..t_2$ from D';
(2)   let R_i be one of the solutions returned by $SAT_{\mathcal{FD}}(D' \wedge \text{labeling}([X]))$;
(3)   let $\langle S'', D''\rangle$ be one of the solutions returned by $SAT_{\mathcal{SET}+\mathcal{FD}}(S' \wedge \tau^{-1}(R_i))$;
        if no solution exists then **return** false;
        else $S' := S' \wedge S''$; $D' := D' \wedge D''$
    end repeat;
    return $S' \wedge \tau^{-1}(D')$

Figure 4: Global Constraint Solver

4.5 Constraint Solving Procedures

The overall structure of the STEP procedure is illustrated in Figure 5. The procedure successively applies specialized constraint rewriting procedures, each processing a different type of primitive constraints.

The different constraint rewriting procedures follow the same spirit of the rewriting procedures for CLP($\mathcal{SET}$) described in [12]. These procedures have to be appropriately modified to include the interaction with the CLP($\mathcal{FD}$) solver, by properly distributing the constraints between the two solvers.

The different rewriting procedures share a common structure. Each one takes as input a pair $\langle S, D\rangle$. Different rewriting rules are applied to S to propagate knowledge to the other constraints. Each rule is selected according to the syntax of the corresponding primitive constraint. During the rewriting process, constraints that might be managed by $SAT_{\mathcal{FD}}$ are separately collected (in D) and ultimately passed to $SAT_{\mathcal{FD}}$—by calling $SAT_{\mathcal{FD}}(D)$. The resulting simplified constraint (e.g., reduced through arc and bound consistency) is then added back to S using the reverse $\rho(D)$ operation. This has the effect of propagating the simplifications performed by $SAT_{\mathcal{FD}}$ to the set constraints.

Remark 13 *The $\mathcal{FD}$-constraints could be simply accumulated in the CLP($\mathcal{FD}$) constraint store, without solving them during the $SAT_{\mathcal{SET}}$ computation. Then, the CLP($\mathcal{FD}$) constraint store could be processed by $SAT_{\mathcal{FD}}$ at the end, after the termination of $SAT_{\mathcal{SET}}$. However, interleaving $SAT_{\mathcal{FD}}$ and $SAT_{\mathcal{SET}}$ activities, as done in the resolution procedures discussed below, practically provides a more effective pruning of the solution space.*

In the rest of this Section, we describe the various rewriting procedures used by $SAT_{\mathcal{SET}}$. Each procedure is presented in its entirety, though the discussion will mostly focus on the changes made w.r.t. the corresponding procedures presented in [12] to allow communication of constraints with the CLP($\mathcal{FD}$) solver.

Procedure $\mathsf{STEP}(\langle S, D\rangle)$:
$\langle S, D\rangle := \mathsf{set\_solve}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{integer\_solve}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{not\_integer\_solve}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{not\_union}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{not\_disj}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{member}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{union}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{disj}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{not\_member}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{not\_equal}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{equal}(\langle S, D\rangle)$;
$\langle S, D\rangle := \mathsf{less\_equal}(\langle S, D\rangle)$;
return $\langle S, D\rangle$

Figure 5: The procedure STEP

4.5.1 set_int_infer Procedure

The objective of the set_int_infer procedure (Figure 6) is to add to the current constraint all the set and integer constraints that can be inferred from the use of terms in the constraint.

In particular:

- The terms appearing as second sub-term of terms of the type $\{\cdot \mid \cdot\}$ are required to be sets—e.g., in a term $\{s \mid t\}$ the term t is required to be a set;
- The terms appearing in int terms or within terms with functors from F_Z are required to be integers;
- The terms used on the right-hand side of $\in$ or $\notin$-constraints are required to be sets;
- The terms present in constraints of the form $||$, $\not\parallel$, $\cup_3$ and $\not\cup_3$ are required to be sets;
- The terms present in constraints $\leq$ are required to be integers.

Other set and integer constraints will be added to the current constraint by the different constraint rewriting procedures—e.g., to characterize the role of newly generated variables. integer constraints are also generated via static program analysis, as discussed in Section 5. Details on how the constraints based on integer and not_integer are handled, and how they interact with the set constraints, are given in Section 4.5.10.

4.5.2 Management of $\in$-constraints

The procedure which performs rewriting of membership constraints ($\in$-constraints) is illustrated in Figure 7. The cases (1), (3), and (4) are identical to those in [12]. In particular, observe that case (3), used to recursively split a membership between an element and a set term, is non-deterministic—i.e., an element r can be considered a member of $\{s \mid t\}$ if either is equal to s or it belongs to t. Case (2) deals with the case where a membership constraint can be handled by the CLP($\mathcal{FD}$) solver—namely, the set is composed

```
Procedure set_int_infer(C)
R := C;
for each primitive constraint c in C do
    %% let t_1, ..., t_n be the arguments of c
    for i:=1 to n do
        R := R ∧ find_int_set(t_i)
    end_for
    case c of
        t_1 ∈ t_2 or t_1 ∉ t_2: R := R ∧ set(t_2);
        t_1 || t_2 or t_1 ∦ t_2: R := R ∧ set(t_1) ∧ set(t_2);
        ∪_3(t_1, t_2, t_3) or ∪̸_3(t_1, t_2, t_3): R := R ∧ set(t_1) ∧ set(t_2) ∧ set(t_3);
        t_1 ≤ t_2: R := R ∧ integer(t_1) ∧ integer(t_2)
    end case
end for
return R

Procedure find_int_set(t)
    if t is a variable X or t is a constant symbol then
            return true;
    elseif t has the form int(t_1, t_2) then
            return integer(t_1) ∧ integer(t_2);
    elseif t has the form f(t_1, ..., t_n), n > 0, f ∈ F_Z then
            return integer(t_1) ∧ ··· ∧ integer(t_n) ;
    elseif t has the form {t_1, ..., t_n | s} then
            return find_int_set(t_1) ∧ ··· ∧ find_int_set(t_n) ∧ set(s);
    elseif t has the form f(t_1, ..., t_n), n > 0 then
            return find_int_set(t_1) ∧ ··· ∧ find_int_set(t_n);
    endif
```

Figure 6: Inference of set and integer constraints.

of ground integers. Cases (5) and (6) deal with constraints of the type $r \in \mathsf{int}(s,t)$. In both these two cases, as well as in case (2), the constraint is passed to D after applying τ to it.

Note that, thanks to the set_int_infer procedure, if the constraint C contains a constraint of the type $s \in \mathsf{int}(t_i, t_f)$, then it contains also the constraints $\mathsf{integer}(t_i)$ and $\mathsf{integer}(t_f)$. In particular, this ensures that, if one of the end-points of the interval (i.e., t_i or t_f) is a variable, then the variable can be instantiated only with integer values.

4.5.3 Management of $\notin$-constraints

Figure 8 shows the procedure used to handle the constraints based on the $\notin$ predicate. The cases (1), (2), and (3) deal with the basic cases of resolving a non-membership w.r.t. a set. Case (4) is non-deterministic, and it describes the three possible cases that guarantee the non-membership of one element to an interval—i.e., the element r is outside of the interval range or r has a non-integer sort. Case (5) deals with the situation where the right-hand side of the constraint is a variable and the left-hand side is a compound integer term—in which case the $\mathcal{FD}$ solver is employed to simplify the integer term.

4.5.4 Management of $=$-constraints

The procedure equal, presented in Figure 9, is used to manage the constraints based on the $=$ predicate. In particular:

- Cases (1)–(4) are obvious cases—in particular, cases (3) and (4) perform the occur-check test as in traditional unification [21].
- Case (5) makes use of an equation $X = t$ to generate a binding for X; note that the effect of applying a

Procedure $\mathsf{member}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle s \in \emptyset \wedge S', D\rangle$ **then**
    return false
(2) **elseif** $\langle S, D\rangle = \langle s \in \{t_1, \ldots, t_n\} \wedge S', D\rangle$ **and**
      $t_1, \ldots, t_n$ are ground integer terms **then**
    $S := S' \wedge \mathsf{integer}(s)$; $D := D \wedge \tau(s \in \{t_1, \ldots, t_n\})$
(3) **elseif** $\langle S, D\rangle = \langle r \in \{s \,|\, t\} \wedge S', D\rangle$ **then**
    (i) $S := S' \wedge r = s$; $D := D \wedge \tau(r = s)$ **OR**
    (ii) $S := S' \wedge r \in t$
(4) **elseif** $\langle S, D\rangle = \langle t \in X \wedge S', D\rangle$ **then**
    $S := S' \wedge X = \{t \,|\, N\} \wedge \mathsf{set}(N)$
(5) **elseif** $\langle S, D\rangle = \langle r \in \mathsf{int}(s, t) \wedge S', D\rangle$ **and** s, t integer constants **then**
    $S := S' \wedge \mathsf{integer}(r)$; $D := D \wedge \tau(r \in \mathsf{int}(s, t))$
(6) **elseif** $\langle S, D\rangle = \langle r \in \mathsf{int}(s, t) \wedge S', D\rangle$ **and**
      s or t compound integer terms or variables **then**
    $S := S' \wedge \mathsf{integer}(r)$; $D := D \wedge \tau(s \leq r) \wedge \tau(r \leq t)$
  endif
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 7: $\in$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Procedure $\mathsf{not\_member}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle s \notin \emptyset \wedge S', D\rangle$ **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle r \notin \{s \,|\, t\} \wedge S', D\rangle$ **then**
    $S := S' \wedge r \neq s \wedge r \notin t$; $D := D \wedge \tau(r \neq s)$
(3) **elseif** $\langle S, D\rangle = \langle t \notin X \wedge S', D\rangle$ **and** $X \in vars(t)$ **then**
    $S := S'$
(4) **elseif** $\langle S, D\rangle = \langle r \notin \mathsf{int}(s, t) \wedge S', D\rangle$ **then**
    (i) $S := S' \wedge \mathsf{integer}(r)$; $D := D \wedge \tau(r + 1 \leq s)$ **OR**
    (ii) $S := S' \wedge \mathsf{integer}(r)$; $D := D \wedge \tau(t + 1 \leq r)$ **OR**
    (iii) $S := S' \wedge \mathsf{not\_integer}(r)$
(5) **elseif** $\langle S, D\rangle = \langle r \notin X \wedge S', D\rangle$ **and** r is a compound integer term **then**
    $S := S' \wedge N \notin X \wedge \mathsf{integer}(N)$; $D := D \wedge \tau(N = r)$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S' \wedge \rho(D), D\rangle$

Figure 8: $\notin$–constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

substitution is propagated to the $\mathcal{FD}$ part of the constraint as well, by adding the constraint $\tau(X = t)$ to it.

- Case (6) expresses the fact that an equation of the form $X = \{\cdots \mid X\}$, intuitively, represents the problem $X = \{\cdots\} \cup X$, whose most general solution can be represented as $X = \{\cdots \mid N\}$, where N is a brand new variable.
- Cases (7) and (8) are traditional cases of unification between Herbrand compound terms.
- Cases (9) and (10) handle equalities between two sets—unfolding them as equalities between the elements of the sets. Observe that, in the sub-cases (i)–(iii), the new equality between set elements is also communicated to the $\mathcal{FD}$ part of the constraint. Note that, if the equalities are not between integers, then the τ function will effectively not change the $\mathcal{FD}$-constraint.
- Case (11) deals with the equality between two intervals.
- Case (12) deals with the equality between an interval and a set term.
- Case (13) deals with equations involving compound integer terms (i.e., terms of the form $f(t_1, \ldots, t_n)$, $f \in F_Z$), such as, for instance, $X = 2 + Y$, $2 + X = 3 * 2$, $2 + X = 5$. The solution of these equations is delegated to the $SAT_{\mathcal{FD}}$ solver. Note that in this case equations of the form $X = f(\ldots X \ldots)$ with $f \in F_Z$ can admit solutions (e.g., $X = X + 1 - 1$).

Observe also that, due to the high degree of non-determinism and to the fact that substitutions are applied (which typically increases the size of the constraints), in the management of $=$-constraints we favor the application of $SAT_{\mathcal{FD}}$ at each cycle of rewriting, thus increasing the rate of propagation between the two solvers.

4.5.5 Management of $\neq$-constraints

The procedure used to rewrite $\neq$-constraints is presented in Figure 10. The cases (1)–(8) do not deal with intervals and integer terms. Cases (1), (3)–(6) deal with those situations where the inequality is trivially satisfied or unsatisfiable. Case (2), non-deterministically, reduces an inequality between two compound terms to an inequality between arguments of the terms. Case (7) resolves an inequality between a set X and a set that contains X as a subset. Case (8) handles an inequality between two sets, using the traditional extensionality principle. The new cases, w.r.t. the procedure presented in [12], are (9)–(11). Cases (9) and (10) propagate inequalities involving integer terms to $SAT_{\mathcal{FD}}$. Case (11) handles the comparison between an interval and a set. In order for an interval $\mathsf{int}(s_1, t_1)$ to be different from a set $\{s_2 \mid t_2\}$, there must be an element in the interval that does not belong to the set or vice versa. This condition is expressed by the two non-deterministic cases in step (11).

4.5.6 Management of $\cup_3$-constraints

The procedure used to rewrite $\cup_3$-constraints used in $SAT_{\mathcal{SET}+\mathcal{FD}}$ is presented in Figure 11. Cases (1)–(5) deals with cases where one of the arguments of the union is a known set. In particular,

- case (1) handles the simple case of union of two identical sets
- case (2) and (3) deal with unions where one of the arguments is the empty set
- case (4) deals with the situation where the result of the union is a known set, while case (5) assumes knowledge of the elements of one of the sets being unioned.

Cases (6) and (7) propagate knowledge from the union constraint to another, inequality, constraint. The new cases, w.r.t. [12], are (8) and (9) and they deal with unions that involve intervals. Case (8) is analogous to case (4), as it simplifies a $\cup_3$-constraint where the third argument (the result of the union) is known to be an interval—the simplification is aimed at asserting that the leftmost extreme of the interval needs to be present in at least one of the two sets that are unioned. Case (9) is analogous to case (5), and it deals with an interval appearing as one of the first two arguments of the union; the reduction relies on showing that the leftmost extreme of the interval has to be present in the result of the union. Cases (10) and (11) complete cases (8) and (9) respectively, dealing with empty intervals.

Procedure equal($\langle S, D\rangle$)
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle X = X \wedge S', D\rangle$ **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle t = X \wedge S', D\rangle$ **and** t is not a variable **then**
    $S := X = t \wedge S'$
(3) **elseif** $\langle S, D\rangle = \langle X = f(t_1, \ldots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv \{\cdot\,|\,\cdot\}$, $f \notin F_Z$, **and** $X \in vars(t_1, \ldots, t_n)$ **then**
    return false
(4) **elseif** $\langle S, D\rangle = \langle X = \{t_0, \ldots, t_n \,|\, t\} \wedge S', D\rangle$ **and** t is $\emptyset$ or a variable **and** $X \in vars(t_0, \ldots, t_n)$ **then**
    return false
(5) **elseif** $\langle S, D\rangle = \langle X = t \wedge S', D\rangle$ **and** $X \notin vars(t)$ **and** t is not a compound integer term **then**
    $S := X = t \wedge S'[X/t]$; $D := \tau(X = t) \wedge D$
(6) **elseif** $\langle S, D\rangle = \langle X = \{t_0, \ldots, t_n \,|\, X\} \wedge S', D\rangle$ **and** $X \notin vars(t_0, \ldots, t_n)$ **then**
    $S := S' \wedge X = \{t_0, \ldots, t_n \,|\, N\} \wedge \mathsf{set}(N)$ (N a new variable)
(7) **elseif** $\langle S, D\rangle = \langle f(s_1, \ldots, s_m) = g(t_1, \ldots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv g$, $f \notin F_Z$ or $g \notin F_Z$ **then**
    return false
(8) **elseif** $\langle S, D\rangle = \langle f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv \{\cdot\,|\,\cdot\}$, $f \notin F_Z$, $f \not\equiv \mathsf{int}$ **then**
    $S := s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge S'$; $D := \tau(s_1 = t_1) \wedge \cdots \wedge \tau(s_n = t_n) \wedge D$
(9) **elseif** $\langle S, D\rangle = \langle \{t \,|\, s\} = \{t' \,|\, s'\} \wedge S', D\rangle$ **and** $\mathsf{tail}(s), \mathsf{tail}(s')$ are not the same variable **then**
  (i) $S := t = t' \wedge s = s' \wedge S'$; $D := \tau(t = t') \wedge D$; **OR**
  (ii) $S := t = t' \wedge \{t \,|\, s\} = s' \wedge S'$; $D := \tau(t = t') \wedge D$; **OR**
  (iii) $S := t = t' \wedge s = \{t' \,|\, s'\} \wedge S'$; $D := \tau(t = t') \wedge D$; **OR**
  (iv) $S := s = \{t' \,|\, N\} \wedge \{t \,|\, N\} = s' \wedge \mathsf{set}(N) \wedge S'$
(10) **elseif** $\langle S, D\rangle = \langle \{t_0, \ldots, t_m \,|\, X\} = \{t'_0, \ldots, t'_n \,|\, X\} \wedge S', D\rangle$ **and** X is a variable **then**
  (i) $S := t_0 = t'_j \ \wedge \{t_1, \ldots, t_m \,|\, X\} = \{t'_0, \ldots, t'_{j-1}, t'_{j+1}, \ldots, t'_n \,|\, X\} \wedge S'$; $D := \tau(t_0 = t'_j) \wedge D$; **OR**
  (ii) $S := t_0 = t'_j \ \wedge \{t_0, \ldots, t_m \,|\, X\} = \{t'_0, \ldots, t'_{j-1}, t'_{j+1}, \ldots, t'_n \,|\, X\} \wedge S'$; $D := \tau(t_0 = t'_j) \wedge D$; **OR**
  (iii) $S := t_0 = t'_j \ \wedge \{t_1, \ldots, t_m \,|\, X\} = \{t'_0, \ldots, t'_n \,|\, X\} \wedge S'$; $D := \tau(t_0 = t'_j) \wedge D$; **OR**
  (iv) $S := X = \{t_0 \,|\, N\} \ \wedge \{t_1, \ldots, t_m \,|\, N\} = \{t'_0, \ldots, t'_n \,|\, N\} \wedge \mathsf{set}(N) \wedge S'$
(11) **elseif** $\langle S, D\rangle = \langle \mathsf{int}(s_1, s_2) = \mathsf{int}(t_1, t_2) \wedge C', D\rangle$ **then**
  (i) $S := S' \wedge s_1 = t_1 \wedge s_2 = t_2$; $D := D \wedge \tau(s_1 = t_1) \wedge \tau(s_2 = t_2)$; **OR**
  (ii) $S := S'$; $D := D \wedge \tau(s_2 + 1 \le s_1) \wedge \tau(t_2 + 1 \le t_1)$
(12) **elseif** $\langle S, D\rangle = \langle \{s \mid s'\} = \mathsf{int}(t, t') \wedge C', D\rangle$ (or $\mathsf{int}(t, t') = \{s \mid s'\}$) **then**
  (i) $S := S' \wedge \cup_3(\mathsf{int}(t, s), \mathsf{int}(s + 1, t'), s') \wedge \mathsf{integer}(s)$;
    $D := D \wedge \tau(t \le s) \wedge \tau(s \le t')$
  (ii) $S := S' \wedge \cup_3(\mathsf{int}(t, s - 1), \mathsf{int}(s + 1, t'), s') \wedge \mathsf{integer}(s)$;
    $D := D \wedge \tau(t \le s) \wedge \tau(s \le t')$
(13) **elseif** $\langle S, D\rangle = \langle s = t \wedge S', D\rangle$ **and** either s or t are compound integer terms **then**
    $S := S' \wedge \mathsf{integer}(s) \wedge \mathsf{integer}(t)$; $D := D \wedge \tau(s = t)$
  end if;
  $D := SAT_{\mathcal{FD}}(D)$;
  if $D = \mathsf{false}$ **then**
    return false
  else
    $S := S' \wedge \rho(D)$
end repeat;
return $\langle S, D\rangle$

Figure 9: =-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Procedure $\mathsf{not\_equal}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle f(s_1, \dots, s_m) \neq g(t_1, \dots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv g$, $f \notin F_Z$ or $g \notin F_Z$ **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle f(s_1, \dots, s_n) \neq f(t_1, \dots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv \{\cdot \mid \cdot\}, f \notin F_Z, n > 0$ **then**
  (i) $S := s_1 \neq t_1 \wedge S'$; $D := \tau(s_1 \neq t_1) \wedge D$ **OR**
    $\vdots$
  (n) $S := s_n \neq t_n \wedge S'$; $D := \tau(s_n \neq t_n) \wedge D$
(3) **elseif** $\langle S, D\rangle = \langle s \neq s \wedge S', D\rangle$ **and** s is a constant or a variable **then**
    return false
(4) **elseif** $\langle S, D\rangle = \langle t \neq X \wedge S', D\rangle$ **and** t is not a variable **then**
    $S := X \neq t \wedge S'$
(5) **elseif** $\langle S, D\rangle = \langle X \neq f(t_1, \dots, t_n) \wedge S', D\rangle$ **and** $f \not\equiv \{\cdot \mid \cdot\}, f \notin F_Z, X \in \mathit{vars}(t_1, \dots, t_n)$ **then**
    $S := S'$
(6) **elseif** $\langle S, D\rangle = \langle X \neq \{t_1, \dots, t_n \mid t\} \wedge S', D\rangle$ **and** $X \in \mathit{vars}(t_1, \dots, t_n)$ **then**
    $S := S'$
(7) **elseif** $\langle S, D\rangle = \langle X \neq \{t_1, \dots, t_n \mid X\} \wedge S', D\rangle$ **and** $X \notin \mathit{vars}(t_1, \dots, t_n)$ **then**
  (i) $S := t_1 \notin X \wedge S'$ **OR**
    $\vdots$
  (n) $S := t_n \notin X \wedge S'$
(8) **elseif** $\langle S, D\rangle = \langle \{s \mid r\} \neq \{u \mid t\} \wedge S', D\rangle$ **then**
  (i) $S := N \in \{s \mid r\} \wedge N \notin \{u \mid t\} \wedge S'$ **OR**
  (ii) $S := N \notin \{s \mid r\} \wedge N \in \{u \mid t\} \wedge S'$
(9) **elseif** $\langle S, D\rangle = \langle X \neq t \wedge S', D\rangle$ **and** t is an integer constant or a variable **then**
    $D := \tau(X \neq t) \wedge D$
(10) **elseif** $\langle S, D\rangle = \langle s \neq t \wedge S', D\rangle$ **and** either s or t are compound integer terms **then**
    $S := S' \wedge \mathsf{integer}(s) \wedge \mathsf{integer}(t)$; $D := \tau(s \neq t) \wedge D$
(11) **elseif** $\langle S, D\rangle = \langle \mathsf{int}(s_1, t_1) \neq \{s_2 \mid t_2\} \wedge S', D\rangle$ (or $\{s_2 \mid t_2\} \neq \mathsf{int}(s_1, t_1)$) **then**
  (i) $S := S' \wedge N \in \mathsf{int}(s_1, t_1) \wedge N \notin \{s_2 \mid t_2\} \wedge \mathsf{integer}(N)$ **OR**
  (ii) $S := S' \wedge N \notin \mathsf{int}(s_1, t_1) \wedge N \in \{s_2 \mid t_2\}$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 10: $\neq$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Procedure $\mathsf{union}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \cup_3(s, s, t) \wedge S', D\rangle$ **then**
    $S := s = t \wedge S'$
(2) **elseif** $\langle S, D\rangle = \langle \cup_3(s, t, \emptyset), D\rangle$ **and** $s \not\equiv t$ **then**
    $S := s = \emptyset \wedge t = \emptyset \wedge S'$
(3) **elseif** $\langle S, D\rangle = \langle \cup_3(\emptyset, t, X) \wedge S', D\rangle$ (or $\cup_3(t, \emptyset, X)$) **and** $t \not\equiv \emptyset$ **then**
    $S := X = t \wedge S'$
(4) **elseif** $\langle S, D\rangle = \langle \cup_3(s_1, s_2, \{t_1 \mid t_2\}) \wedge S', D\rangle$ **and** $s_1 \not\equiv s_2$ **then**
    $S := S' \wedge \{t_1 \mid t_2\} = \{t_1 \mid N\} \wedge t_1 \notin N \wedge t_1 \notin N_1 \wedge \mathsf{set}(N) \wedge \mathsf{set}(N_1)$;
  (i) $S := S \wedge s_1 = \{t_1 \mid N_1\} \wedge \cup_3(N_1, s_2, N)$ **OR**
  (ii) $S := S \wedge \{t_1 \mid t_2\} = \{t_1 \mid N\} \wedge s_2 = \{t_1 \mid N_1\} \wedge \cup_3(s_1, N_1, N)$ **OR**
  (iii) $S := S \wedge s_1 = \{t_1 \mid N_1\} \wedge s_2 = \{t_1 \mid N_2\} \wedge t_1 \notin N_2 \wedge \mathsf{set}(N_2) \wedge \cup_3(N_1, N_2, N)$
(5) **elseif** $\langle S, D\rangle = \langle \cup_3(\{t_1 \mid t_2\}, t, X) \wedge S', D\rangle$ (or $\cup_3(t, \{t_1 \mid t_2\}, X)$) **and** $t \not\equiv \{t_1 \mid t_2\}, t \not\equiv \emptyset$ **then**
    $S := S' \wedge \{t_1 \mid t_2\} = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge \mathsf{set}(N) \wedge \mathsf{set}(N_1)$;
  (i) $S := S \wedge t_1 \notin t \wedge \cup_3(N_1, t, N)$ **OR**
  (ii) $S := S \wedge t = \{t_1 \mid N_2\} \wedge \mathsf{set}(N_2) \wedge t_1 \notin N_2 \wedge \cup_3(N_1, N_2, N)$
(6) **elseif** $\langle S, D\rangle = \langle \cup_3(X, Y, Z) \wedge Z \neq t \wedge S', D\rangle$ **and** $X \not\equiv Y$ **then**
  (i) $S := S' \wedge \cup_3(X, Y, Z) \wedge N \in Z \wedge N \notin t$ **OR**
  (ii) $S := S' \wedge \cup_3(X, Y, Z) \wedge N \in t \wedge N \notin Z$ **OR**
  (iii) $S := S' \wedge \cup_3(X, Y, Z) \wedge Z = \emptyset \wedge t \neq \emptyset$
(7) **elseif** $\langle S, D\rangle = \langle \cup_3(X, Y, Z) \wedge X \neq t \wedge S', D\rangle$ (or $\cup_3(Y, X, Z) \wedge X \neq t$) **and** $X \not\equiv Y$ **then**
  (i) $S := S' \wedge \cup_3(X, Y, Z) \wedge N \in X \wedge N \notin t$ **OR**
  (ii) $S := S' \wedge \cup_3(X, Y, Z) \wedge N \in t \wedge N \notin X$ **OR**
  (iii) $S := S' \wedge \cup_3(X, Y, Z) \wedge X = \emptyset \wedge t \neq \emptyset$
(8) **elseif** $\langle S, D\rangle = \langle \cup_3(s_1, s_2, \mathsf{int}(t_1, t_2)) \wedge S', D\rangle$ **and** t_1, t_2 are ground **and** $t_1 \leq t_2$ **then**
  (i) $S := S' \wedge s_1 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \mathsf{set}(N_1) \wedge \cup_3(N_1, s_2, \mathsf{int}(t_1 + 1, t_2))$ **OR**
  (ii) $S := S' \wedge s_2 = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge \mathsf{set}(N_1) \wedge \cup_3(s_1, N_1, \mathsf{int}(t_1 + 1, t_2))$ **OR**
  (iii) $S := S' \wedge s_1 = \{t_1 \mid N_1\} \wedge s_2 = \{t_1 \mid N_2\} \wedge t_1 \notin N_1 \wedge t_1 \notin N_2 \wedge$
      $\mathsf{set}(N_1) \wedge \mathsf{set}(N_2) \wedge \cup_3(N_1, N_2, \mathsf{int}(t_1 + 1, t_2))$
(9) **elseif** $\langle S, D\rangle = \langle \cup_3(\mathsf{int}(t_1, t_2), s, X) \wedge S', D\rangle$ (or $\cup_3(s, \mathsf{int}(t_1, t_2), X)$) **and** t_1, t_2 are ground **and** $t_1 \leq t_2$ **then**
  (i) $S := S' \wedge X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge t_1 \notin s \wedge \mathsf{set}(N) \wedge \cup_3(\mathsf{int}(t_1 + 1, t_2), s, N)$ **OR**
  (ii) $S := S' \wedge X = \{t_1 \mid N\} \wedge t_1 \notin N \wedge s = \{t_1 \mid N_1\} \wedge t_1 \notin N_1 \wedge$
      $\mathsf{set}(N) \wedge \mathsf{set}(N_1) \wedge \cup_3(\mathsf{int}(t_1 + 1, t_2), N_1, N)$
(10) **elseif** $\langle S, D\rangle = \langle \cup_3(s_1, s_2, \mathsf{int}(t_1, t_2)) \wedge S', D\rangle$ **and** t_1, t_2 are ground **and** $t_1 > t_2$ **then**
    $S := S' \wedge s_1 = \emptyset \wedge s_2 = \emptyset$
(11) **elseif** $\langle S, D\rangle = \langle \cup_3(\mathsf{int}(t_1, t_2), s, X) \wedge S', D\rangle$ (or $\cup_3(s, \mathsf{int}(t_1, t_2), X)$) **and** t_1, t_2 are ground **and** $t_1 > t_2$ **then**
    $S := S' \wedge X = s$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 11: $\cup_3$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

4.5.7 Management of $\not\emptyset_3$-constraints

The procedure in Figure 12 illustrates the handling of the $\not\emptyset_3$-constraints. The procedure is identical to the procedure presented in [12], and it relies on the extensionality principle of set equality—i.e., either there is one element in the result of the union that does not appear in neither of the two sets being unioned, or there is an element in one of the sets being unioned that does not appear in the result of the union.

Procedure not_union($\langle S, D\rangle$)
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \not\emptyset_3(s_1, s_2, s_3) \wedge S', D\rangle$ **then**
  (i) $S := S' \wedge N \in s_3 \wedge N \notin s_1 \wedge N \notin s_2$
  (ii) $S := S' \wedge N \in s_1 \wedge N \notin s_3$
  (iii) $S := S' \wedge N \in s_2 \wedge N \notin s_3$
  end if
end repeat;
return $\langle S, D\rangle$

Figure 12: $\not\emptyset_3$-constraints

4.5.8 Management of $||$-constraints and $\not||$-constraints

The procedure which handles the $||$-constraints is presented in Figure 13. Cases (1) and (2) handle situations where the constraint is solvable in a straightforward manner. Case (3) and (4) handle disjoint constraints in presence of set terms. Cases (5) and (6) deal with the presence of intervals. The first one deals with a request of disjointness between two intervals, which is resolved by generating appropriate inequalities to be handled by $SAT_{\mathcal{FD}}$. Case (6) deals with other disjointness cases dealing with intervals—which are resolved by progressively unfolding the interval.

The handling of the $\not||$-constraints is performed as illustrated in Figure 14. The only new case is case (1) which is symmetrical to the case (5) of the procedure for the $||$-constraints. Case (2) resolves the constraint by requiring the existence of an element in common between the two arguments.

4.5.9 Management of $\leq$-constraints

The rewriting rules for $\leq$-constraints used in $SAT_{\mathcal{SET}+\mathcal{FD}}$ are defined in Figure 15. The procedure simply extracts the $\leq$-constraints and delivers them to the $SAT_{\mathcal{FD}}$ solver.

4.5.10 Management of set, integer and not_integer constraints

The procedure in Figure 16 is employed to handle the set-constraints. The cases (1)–(3) handle the possible cases where the shape of the argument is such to allow us to simplify the constraints (e.g., the argument is known to be a set or an interval). In case (4), we are detecting a "type conflict", while case (5) is used to remove unnecessary not_integer-constraints.

The procedures in Figure 17 and 18 perform a similar transformation for the integer- and not_integer-constraints. Intuitively, an integer-constraint succeeds on integer constants and fails on any non-variable non-integer term. Similarly for the not_integer-constraint which also needs to interact with the integer-constraint (e.g., integer(X) fails if there is a not_integer(X) in the constraint).

4.6 Soundness and Completeness

In Section 3.2 we presented a semantics for the CLP($\mathcal{SET}, \mathcal{FD}$) language. The objective of the following results is to show that $SAT_{\mathcal{SET}+\mathcal{FD}}$ is a sound and complete solver with respect to the selected set structure $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}}$.

Procedure $\mathsf{disj}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \emptyset \ || \ t \wedge S', D\rangle$ (or $t \ || \ \emptyset$) **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle X \ || \ X \wedge S', D\rangle$ **then**
    $S := X = \emptyset \wedge S'$
(3) **elseif** $\langle S, D\rangle = \langle \{t_1 \,|\, t_2\} \ || \ X \wedge S', D\rangle$ (or $X \,||\, \{t_1 \,|\, t_2\}$) **then**
    $S := t_1 \notin X \wedge X \ || \ t_2 \wedge S'$
(4) **elseif** $\langle S, D\rangle = \langle \{t_1 \,|\, s_1\} \ || \ \{t_2 \,|\, s_2\} \wedge S', D\rangle$ **then**
    $S := S' \wedge t_1 \neq t_2 \wedge t_1 \notin s_2 \wedge t_2 \notin s_1 \wedge s_1 \,||\, s_2$
    $D := D \wedge \tau(t_1 \neq t_2)$
(5) **elseif** $\langle S, D\rangle = \langle \mathsf{int}(s_1, s_2) \ || \ \mathsf{int}(t_1, t_2) \wedge S', D\rangle$ **then**
    $S := S'$
 (i) $D := D \wedge \tau(s_2 + 1 \leq t_1)$ **OR**
 (ii) $D := D \wedge \tau(t_2 + 1 \leq s_1)$ **OR**
 (iii) $D := D \wedge \tau(s_2 + 1 \leq s_1) \wedge \tau(t_2 + 1 \leq t_1)$
(6) **elseif** $\langle S, D\rangle = \langle \mathsf{int}(s_1, s_2) \ || \ t \wedge S', D\rangle$ (or $t \ || \ \mathsf{int}(s_1, s_2)$) **and** s_1, s_2 are ground **then**
 (i) $S := S' \wedge s_1 \notin t \wedge \mathsf{int}(s_1 + 1, s_2) \ || \ t$; $D := D \wedge \tau(s_1 + 1 \leq s_2)$ **OR**
 (ii) $S := S'$; $D := D \wedge \tau(s_2 + 1 \leq s_1)$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 13: $||$-constraints

Procedure $\mathsf{not\_disj}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \mathsf{int}(s_1, s_2) \not\parallel \mathsf{int}(t_1, t_2) \wedge S', D\rangle$ **then**
 (i) $S := S'$; $D := D \wedge \tau(s_1 \leq t_1) \wedge \tau(t_1 \leq s_2)$ **OR**
 (ii) $S := S'$; $D := D \wedge \tau(t_1 \leq s_1) \wedge \tau(s_1 \leq t_2)$
(2) **elseif** $\langle S, D\rangle = \langle s \not\parallel t \wedge S', D\rangle$ **then**
    $S := S' \wedge N \in s \wedge N \in t$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else **return** $\langle S \wedge \rho(D), D\rangle$

Figure 14: $\not\parallel$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Procedure less_equal($\langle S, D\rangle$)
repeat until (no cases apply)
  if $\langle S, D\rangle =$ false **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle s \leq t \wedge S', D\rangle$ **and** s, t are integer terms or variables
    $S := S'$; $D := \tau(s \leq t) \wedge D$
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D =$ false **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 15: $\leq$-constraint rewriting in $SAT_{\mathcal{SET}+\mathcal{FD}}$

Procedure set_solve($\langle S, D\rangle$)
repeat until (no cases apply)
  if $\langle S, D\rangle =$ false **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \mathsf{set}(\emptyset) \wedge S', D\rangle$ **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle \mathsf{set}(\{s \,|\, t\}) \wedge S', D\rangle$ **or** $\langle S, D\rangle = \langle \mathsf{set}(\mathsf{int}(s, t)) \wedge S', D\rangle$ **then**
    $S := S'$
(3) **elseif** $\langle S, D\rangle = \langle \mathsf{set}(f(t_1, \ldots, t_n)) \wedge S', D\rangle$ **then**
    return false
(4) **elseif** $\langle S, D\rangle = \langle \mathsf{set}(X) \wedge \mathsf{integer}(X) \wedge S', D\rangle$ **then**
    return false
(5) **elseif** $\langle S, D\rangle = \langle \mathsf{set}(X) \wedge \mathsf{not\_integer}(X) \wedge S', D\rangle$ **then**
    $S := S' \wedge \mathsf{set}(X)$
  end if
end repeat;
return $\langle S, D\rangle$

Figure 16: set-constraints

Procedure integer_solve($\langle S, D\rangle$)
repeat until (no cases apply)
  if $\langle S, D\rangle =$ false **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \mathsf{integer}(t) \wedge S', D\rangle$ **and** $t \in Z$ **then**
    $S := S'$
(2) **elseif** $\langle S, D\rangle = \langle \mathsf{integer}(f(t_1, \ldots, t_n)) \wedge S', D\rangle$ **and** $f \in F_Z$ **then**
    $S := S' \wedge \mathsf{integer}(t_1) \wedge \ldots \wedge \mathsf{integer}(t_n)$
(3) **elseif** $\langle S, D\rangle = \langle \mathsf{integer}(f(t_1, \ldots, t_n)) \wedge S', D\rangle$ **then**
    return false
  end if
end repeat;
return $\langle S, D\rangle$

Figure 17: integer-constraints

```
    Procedure not_integer_solve(⟨S, D⟩)
    repeat until (no cases apply)
        if ⟨S, D⟩ = false then
            return false
(1)     elseif ⟨S, D⟩ = ⟨not_integer(t) ∧ S', D⟩ and t ∈ Z then
            return false
(2)     elseif ⟨S, D⟩ = ⟨not_integer(f(t_1, ..., t_n)) ∧ S', D⟩ and f ∈ F_Z then
            return false
(3)     elseif ⟨S, D⟩ = ⟨not_integer(f(t_1, ..., t_n)) ∧ S', D⟩ then
            S := S'
(4)     elseif ⟨S, D⟩ = ⟨not_integer(X) ∧ integer(X) ∧ S', D⟩ then
            return false
        end if
    end repeat;
    return ⟨S, D⟩
```

Figure 18: not_integer-constraints

Theorem 14 *(Soundness and Completeness) Let S be a constraint and $\langle S_1, D_1\rangle, \ldots, \langle S_n, D_n\rangle$ be the constraints obtained from $SAT_{\mathcal{SET}+\mathcal{FD}}(S)$. Then,*

1. *if σ is a valuation of $\langle S_i, D_i\rangle$ and $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \sigma(S_i)$ then $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \sigma(S)$, for all $i = 1, \ldots, n$,*
2. *if σ is a valuation of S and $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \sigma(S)$ then there exists i, $1 \leq i \leq n$, such that σ can be expanded to the variables of $vars(S_i) \setminus vars(S)$ so that it satisfies $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \sigma(S_i)$.*

Proof: The proof relies on showing that each rewriting step leads to constraints that are equisatisfiable to the initial one. The proof also assumes that the procedure $SAT_{\mathcal{FD}}$ satisfies the following property: for each $\mathcal{FD}$-constraint D, D and $SAT_{\mathcal{FD}}(D)$ are equisatisfiable w.r.t. $\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}}$.

Let us denote with $\vec{X}$ the variables in S. The first step of $SAT_{\mathcal{SET}+\mathcal{FD}}$ executes the set_int_infer procedure on S, leading to a new constraint $S \wedge S'$. S' contains constraints of the type set and integer, derived from the use of terms and constraints in S. It is easy to see that

$$\mathcal{A}^{\mathcal{SET}}_{\mathcal{FD}} \models \forall \vec{X}(S \wedge S' \leftrightarrow S)$$

Let us now proceed with proving that each step performed by the procedures used inside STEP maintains equisatisfiability of the input and output constraints. More precisely, we want to show that, if a rewriting step applied to $\langle S, D\rangle$ produces $\langle S', D'\rangle$, then $S \wedge \tau^{-1}(D)$ is equisatisfiable to $S' \wedge \tau^{-1}(D')$. We will focus here exclusively on those steps that have been introduced to handle intervals, since the equisatisfiability for the other steps has already been proved in [12].

member: the new cases are cases (2), (5) and (6). In all cases, the property holds directly because of the equisatisfiability result for $SAT_{\mathcal{FD}}$.

not_member: the only new cases are cases (4) and (5). For case (4), from the interpretation of $\in$ we have that $r \in \mathsf{int}(s,t)$ is true iff $\mathsf{integer}(r)$ is true and $r \in \{s, s+1, \ldots, t\}$. Thus, $r \notin \mathsf{int}(s,t)$ is true iff either $\mathsf{not\_integer}(r)$ is true or $r \notin \{s, s+1, \ldots, t\}$. The results for case (5) follows directly from the initial assumption about $SAT_{\mathcal{FD}}$.

equal: the new cases are case (11) and (12). Correctness of case (11) follows from the fact that the interpretation of intervals $\mathsf{int}(s_1, s_2)$ and $\mathsf{int}(t_1, t_2)$ are the sets $\{s_1, s_1+1, \ldots, s_2\}$ and $\{t_1, t_1+1, \ldots, t_2\}$ respectively. These sets are equal iff $s_1 = t_1$ and $s_2 = t_2$ (case (i)), or both of them are empty (case (ii)). Let us consider case (12). If $\{s \mid s'\} = \mathsf{int}(t, t')$, then it must be that $s \in \mathsf{int}(t, t')$, thus, in particular, $t \leq s$ and $s \leq t'$. This means that the interval $\mathsf{int}(t, t')$ can be split into $s_1 = \mathsf{int}(t, s)$ and $s_2 = \mathsf{int}(s+1, t')$. This split is performed by the union constraint. In sub-case (i) we assume that $s \in s'$; in sub-case (ii) that $s \notin s'$. For the other direction, it is immediate to see that satisfiability of all these constraints in (i) or in (ii) implies that $\{s \mid s'\} = \mathsf{int}(t, t')$. Case (13) follows from the correctness of the $SAT_{\mathcal{FD}}$ solver.

<u>not_equal</u>: the new cases are (9)–(11). Cases (9) and (10) simply send redundant information to the CLP($\mathcal{FD}$) solver. Case (11) is nothing but the application of the extensionality principle.

<u>union:</u> The new cases are (8), (9), (10) and (11). For case (8), let us assume that $s_1 \cup s_2 = \mathsf{int}(t_1, t_2)$ and $t_1 \leq t_2$ (if $t_1 > t_2$ then $s_1 = s_2 = \emptyset$ — case (10)). This means that t_1 belongs to s_1 or to s_2. If it belongs to s_1 but not to s_2, let N_1 be the set obtained by removing t_1 from s_1. It holds that $N_1 \cup s_2 = \mathsf{int}(t_1 + 1, t_2)$. This is sub-case (i). Sub-case (ii) is symmetrical. If t_1 belongs to both s_1 and s_2, let N_1 and N_2 be the set obtained by removing t_1 from s_1 and s_2, respectively. Then it holds that $N_1 \cup N_2 = \mathsf{int}(t_1 + 1, t_2)$. This is sub-case (iii). It is immediate to see that if any of the three constraints (i)–(iii) are satisfied, then the same holds for $\cup_3(s_1, s_2, \mathsf{int}(t_1, t_2))$. The proof for cases (9) and (11) follows the same schema.

<u>not_union:</u> it is the same as in [12].

<u>disj:</u> The new cases are (5) and (6). For case (5), it is immediate to see that two intervals $\mathsf{int}(s_1, s_2)$ and $\mathsf{int}(t_1, t_2)$ are disjoint iff $s_2 < t_1$ or $t_2 < s_1$ or both the intervals are empty. Case (6)(i) is similar to case (3), while case (6)(ii) is analogous to case (2).

<u>not_disj</u>: The unique new case is (1) which is the negation of case (5) of disj.

<u>less_equal</u>: This procedure has a unique step that communicates a constraint to $SAT_{\mathcal{FD}}$ and thus its correctness follows from that of $SAT_{\mathcal{FD}}$.

<u>set_solve:</u> Cases (1)–(2) remove a set constraint that is trivially satisfied, while (3) ad (4) detects contradiction. Case (5) removes the not_integer constraint that is implied by the set constraint.

<u>integer_solve:</u> Case (1) removes an integer constraint that is trivially satisfied, case (2) does the same, and moreover, it propagates the integer constraint to arguments. Case (3) detects failure.

<u>not_integer_solve</u>: Cases (1)–(2) are the negation of (1)–(2) of integer_solve. Case (3) removes a constraint that is trivially satisfied, case (4) detects the contradiction of being integer and non-integer at the same time. □

Another concern we should have is the termination of the procedure on the input constraints. We can state the following termination result:

Theorem 15 *The $SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure can be implemented in such a way that it terminates for every input constraint C.*

Proof: [Sketch] Termination follows from the termination of the corresponding procedure of [12] and from the termination of the $SAT_{\mathcal{FD}}$ procedure. The outcome of $SAT_{\mathcal{FD}}$ contains only variables assignments and reduced intervals—and neither of the two can endanger the termination of the main $SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure. □

The termination of $SAT_{\mathcal{SET}+\mathcal{FD}}$ and the finiteness of the number of non-deterministic choices generated during its computation guarantee the finiteness of the number of constraints non-deterministically returned by $SAT_{\mathcal{SET}+\mathcal{FD}}$. The following results extend the previous theorem to prove termination of the global constraint solving procedure described in Figure 4.

Theorem 16 *The Global_$SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure terminates for every input constraint C.*

Proof: [Sketch] The termination of the procedure $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ depends on the termination of the $SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure—argued in Theorem 15—and on the termination of the **repeat** loop. In the loop there are two non-deterministic steps: step (2) and step (3). The first of them generates a finite set of choice points, since the labeling procedure non-deterministically assigns a bounded-domain value to the variable X. The other step generates a finite set of choice points since, for each choice of R_i, the $SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure is called. It follows that the finiteness of non-deterministic choices limits the number of constraints that are non-deterministically returned by the $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure. □

Theorem 17 *Let C be a constraint. If $\mathcal{V}_\in = vars(C)$, then*

1. *if C is satisfiable, then the procedure $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ non-deterministically returns the collection of solved form constraints $\langle S_1, D_1\rangle, \ldots, \langle S_n, D_n\rangle$;*
2. *if C is not satisfiable, then the procedure $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ returns* false.

Proof: [Sketch] As noted in Remark 12, the $SAT_{\mathcal{SET}+\mathcal{FD}}$ solver can return constraints in non-solved form of the following two types:
- $\mathsf{int}(s,t) \,||\, Y$, where s or t (or both) are non-ground, and
- $\cup_3(u,v,z)$, where any of u, v, z are of the form $\mathsf{int}(s,t)$, with s or t (or both) non-ground integer terms, and the remaining of u, v, and z are variables.

We can show that the $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ solver is able to process these constraints and reduce them to a solved form. Referring to Figure 4, let $\langle S', D'\rangle$ be one of the solutions returned by $SAT_{\mathcal{SET}+\mathcal{FD}}(C)$. Let us show that the eventual presence of a constraint in non-solved form is eliminated by the application of the various steps of the $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ procedure.

- $\mathsf{int}(s,t) \,||\, Y \in S'$. We show the result assuming that s is non-ground and t is ground. The other cases are equivalent. Let $V_s = vars(s)$. By hypothesis $V_s \subseteq V_\in$ and for each variable $X \in V_s$, it exists a constraint $X \in t_1..t_2$ in D', with t_1, t_2 ground integers.
  The **repeat** loop selects all $X \in t_1..t_2$ constraints in D', and thus every variable in V_s is considered. For each them R_i is a solution returned by $SAT_{\mathcal{FD}}(D' \wedge \mathsf{labeling}([X]))$. Since $\mathsf{labeling}([\mathrm{X}])$ selects non-deterministically a value for X out of its domain $[t_1..t_2]$, the mapping $\rho(R_i)$ contains the literal $X = n_X$, for every $X \in V_s$, for some integer n_X. Thus, the term s can be reduced to a ground term by means of the application of $SAT_{\mathcal{SET}+\mathcal{FD}}(S' \wedge \rho(R_i))$. Technically, a substitution of $X = n_X$ is required for every occurrence of X as well as a final evaluation of the ground (compound) integer s by means of $SAT_{\mathcal{FD}}$ solver. When s, t are ground integers, $\mathsf{int}(s,t)$ can be rewritten by means of rule (6) of disj procedure, and the original constraint in non-solved form is consumed.
  If the non-deterministic applications of steps (2) and (3) of $\mathsf{Global\_}SAT_{\mathcal{SET}+\mathcal{FD}}$ produce a failure, there is no combination of assignments for the variables V_s from the corresponding domains that is able to satisfy C. Hence, C is not satisfiable and the solver returns false.
- $\cup_3(u,v,z)$. The proof is similar to the previous one. The key idea is again that the intervals can be first determined after labeling and then removed using the rules (8) and (9) of union procedure.

□

4.7 An Extension: size Constraint

The availability of the $\leq$-constraint allows us to introduce new expressive constraints in $\mathcal{SET}$. In particular, it becomes possible to provide a general cardinality constraint size in the context of CLP($\mathcal{SET}$): the constraint $\mathsf{size}(s,h)$ is satisfied if s is a set, h is a non-negative integer, and h corresponds to the cardinality of s.

First of all, the definition of solved form for a CLP($\mathcal{SET}, \mathcal{FD}$)-constraint S must be properly modified by adding the new case:

(vii) $\mathsf{size}(X,N)$ and there are no other literals of the form $\mathsf{size}(X,M)$, with M distinct from N, in S.

The constraint size can be handled using the rewriting rules in Figure 19. Case (1) handles the situation where s is the empty set, while case (2) deals with the situation where s is an interval. Case (3) handles constraints of the type $\mathsf{size}(\{s \mid t\}, h)$:

- in (i) we assume that $s \in t$,
- in (ii) we assume that $s \notin t$.

In all these cases, constraints over h are passed to D after applying τ to them. Finally, case (4) requires the size of a set to be unique.

The procedure set_int_infer needs to be extended to properly account for the new constraint. Specifically, an additional case is introduced, which adds the constraint

$$\mathsf{set}(s) \wedge \mathsf{integer}(h)$$

for each constraint $\mathsf{size}(s, h)$. In addition, the case dealing with $\cup_3$ in set_int_infer is modified so that, each time a constraint of the form $\cup_3(r, s, t)$ is encountered, the constraints

$$\mathsf{size}(r, N_r) \wedge \mathsf{size}(s, N_s) \wedge \mathsf{size}(t, N_t) \wedge N_r \leq N_t \wedge N_s \leq N_t$$

are also introduced. This allows, for instance, the CLP($\mathcal{SET}, \mathcal{FD}$) solver to detect that the constraint

$$R \subseteq S \wedge S \subseteq R \wedge R \neq S$$

is false.

Example 18 *The following CLP($\mathcal{SET}, \mathcal{FD}$)-goals illustrate applications of the* size *constraint.*

:– size({a,b,c,b},N) *returns* N = 3*;*

:– size({X,Y},N) *returns either* N = 2, X $\neq$ Y *or* N = 1, X = Y*;*

:– size(S,2) *returns* S={X,Y}, X $\neq$ Y*;*

:– size({a | R},2) *returns either* R = {X}, X $\neq$ a *or* R = {a,X}, X $\neq$ a.

□

Effective solutions have been proposed to handle *cardinality* constraints for CSP, e.g., [24, 23], and in the context of CLP($\mathcal{FD}$) [29, 4]. This kind of constraints 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 simultaneously instantiated to d. It is therefore a constraint associated to the availability of a fixed number k of resources of a certain kind d.

5 Domain Information Using Static Analysis for CLP($\mathcal{SET}, \mathcal{FD}$)

The procedures in the previous section are capable of distributing primitive constraints between the different solvers. Nevertheless, it is possible to further improve the communication between solvers by *statically* extracting from the programs specific semantical properties, and using such information to guide this distribution process.

5.1 Analysis of Terms and Constraints

In this section we introduce a static analysis scheme for CLP($\mathcal{SET}, \mathcal{FD}$) constraints, aimed at capturing the following property of a term s: s is a nested set of integers and $\mathsf{ni\_set}(s)$ is a measure of the depth of nesting of the sets. The formal 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 $\mathsf{ni\_set}(t) = n$;

Procedure $\mathsf{size\_solve}(\langle S, D\rangle)$
repeat until (no cases apply)
  if $\langle S, D\rangle = \mathsf{false}$ **then**
    return false
(1) **elseif** $\langle S, D\rangle = \langle \mathsf{size}(\emptyset, h) \wedge S', D\rangle$ **then**
    $D := D \wedge \tau(h = 0)$
(2) **elseif** $\langle S, D\rangle = \langle \mathsf{size}(\mathsf{int}(s,t), h) \wedge S', D\rangle$ **then**
    $D := D \wedge \tau(h = t - s + 1)$
(3) **elseif** $\langle S, D\rangle = \langle \mathsf{size}(\{s \mid t\}, h) \wedge S', D\rangle$ **then**
    $D := D \wedge \tau(h \geq 0 \wedge K = h - 1)$;
(i) $S := S' \wedge t = \{s \mid N\} \wedge \mathsf{set}(N) \wedge s \notin N \wedge \mathsf{integer}(K) \wedge \mathsf{size}(N, K)$ **OR**
(ii) $S := S' \wedge s \notin t \wedge \mathsf{integer}(K) \wedge \mathsf{size}(t, K)$
(4) **elseif** $\langle S, D\rangle = \langle \mathsf{size}(X, h) \wedge \mathsf{size}(X, m) \wedge S', D\rangle$ **then**
    $S := S' \wedge \mathsf{size}(X, h) \wedge h = m$;
  end if
end repeat;
$D := SAT_{\mathcal{FD}}(D)$;
if $D = \mathsf{false}$ **then**
  return false
else return $\langle S \wedge \rho(D), D\rangle$

Figure 19: size constraints

- $\mathsf{ni\_set}(s) = \bot$ is non ground;
- $\mathsf{ni\_set}(s) = \top$ otherwise.

We allow the additional values for $\mathsf{ni\_set}(s)$: $\bot$, when no information is currently available for s, and $\top$, when s is not a "regular" nested set of integers (e.g., $\{\{1\}, 1\}$). 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., $\{1, 2, 3, 5\}$).

Our objective is to statically determine the values of ni_set for the various terms in the program and in the goal, and use such information to generate additional constraints (of the type set and integer).

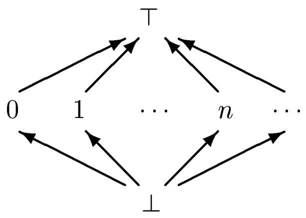

Figure 20: The lattice $\mathcal{L}$

The values for $\mathsf{ni\_set}(s)$ are elements of the lattice $\mathcal{L} = \langle\{\bot, \top, 0, 1, 2, \ldots\}, \prec\rangle$, graphically depicted in Figure 20. The join operator $\sqcup$ is naturally defined on $\mathcal{L}$, as shown in Figure 21.

We introduce two more operations on the lattice $\mathcal{L}$, respectively denoted by "$\oplus$" and "$\ominus$", defined in Figure 21, 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 $\mathcal{L}$.

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($\mathcal{SET}, \mathcal{FD}$) constraint C under consideration. The rules in Figure 22 are applied to the literals and the term of C 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.

For example, from $A + B$ one can infer that $\mathsf{ni\_set}(A) = \mathsf{ni\_set}(A) \sqcup 0$ and $\mathsf{ni\_set}(B) = \mathsf{ni\_set}(B) \sqcup 0$. Similarly, the presence of interval definitions with variables as endpoints, as in $X \in \mathsf{ni\_set}(1, N)$ allows us to

$x \sqcup x = x$	
$x \sqcup y = y \sqcup x$	
$\bot \sqcup x = x$	$x \neq \bot$
$x \sqcup y = \top$	$x \neq y, x \neq \bot, y \neq \bot$

$\bot \oplus \bot = \bot$
$\bot \oplus n = n \oplus \bot = \bot$
$\top \oplus x = x \oplus \top = \top$
$m \oplus n = m +^{\mathbb{Z}} n$

$\top \ominus x = \top$	
$m \ominus n = m -^{\mathbb{Z}} n$	$m \geq n$
$m \ominus n = \top$	$m < n$
$\bot \ominus \bot = \bot \ominus n = \bot$	
$\bot \ominus \top = \top$	

Figure 21: Semantics of $\sqcup$, $\oplus$, and $\ominus$ on $\mathcal{L}$

infer: $\mathsf{ni\_set}(N) = \mathsf{ni\_set}(N) \sqcup 0$. As another example, let us consider the constraint:

$$M \in \mathsf{int}(1, 10) \wedge N = \{\{M\}, M\} \wedge \cup_3(\{M\}, \emptyset, S).$$

The analysis returns

$$\mathsf{ni\_set}(M) = 0, \mathsf{ni\_set}(S) = 1, \mathsf{ni\_set}(N) = \top.$$

(1)	$\emptyset$	$\mapsto$	$\mathsf{ni\_set}(\emptyset) := \bot$
(2)	$t \in Z$	$\mapsto$	$\mathsf{ni\_set}(t) := 0$
(3)	$\left.\begin{array}{r} f(t_1, \dots, t_k) \\ f \in F_Z \end{array}\right\}$	$\mapsto$	$\mathsf{ni\_set}(f(t_1, \dots, t_k)) := \mathsf{ni\_set}(f(t_1, \dots, t_k)) \sqcup 0;$ $\mathsf{ni\_set}(t_1) := \mathsf{ni\_set}(t_1) \sqcup 0;$ $\vdots$ $\mathsf{ni\_set}(t_k) := \mathsf{ni\_set}(t_k) \sqcup 0$
(4)	$\left.\begin{array}{r} f(t_1, \dots, t_k) \\ f \in K \end{array}\right\}$	$\mapsto$	$\mathsf{ni\_set}(f(t_1, \dots, t_k)) := \top$
(5)	$\mathsf{int}(t_i, t_f)$	$\mapsto$	$\mathsf{ni\_set}(\mathsf{int}(t_i, t_f)) := \mathsf{int}(t_i, t_f)) \sqcup 1;$ $\mathsf{ni\_set}(t_i) := \mathsf{ni\_set}(t_i) \sqcup 0;$ $\mathsf{ni\_set}(t_f) := \mathsf{ni\_set}(t_f) \sqcup 0$
(6)	$\{t \mid s\}$	$\mapsto$	$\mathsf{ni\_set}(\{t \mid s\}) := (\mathsf{ni\_set}(t) \oplus 1) \sqcup \mathsf{ni\_set}(s);$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(\{t \mid s\}) \ominus 1;$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(\{t \mid s\})$
(7)	$s = t$	$\mapsto$	$\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t);$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(s)$
(8)	$t \in s$	$\mapsto$	$\mathsf{ni\_set}(t) := \mathsf{ni\_set}(t) \sqcup (\mathsf{ni\_set}(s) \ominus 1));$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(s) \sqcup (\mathsf{ni\_set}(t) \oplus 1)$
(9)	$\cup_3(r, s, t)$	$\mapsto$	$\mathsf{ni\_set}(r) := \mathsf{ni\_set}(r) \sqcup \mathsf{ni\_set}(s) \sqcup \mathsf{ni\_set}(t);$ $\mathsf{ni\_set}(s) := \mathsf{ni\_set}(r);$ $\mathsf{ni\_set}(t) := \mathsf{ni\_set}(r)$
(10)	$\mathsf{integer}(r)$	$\mapsto$	$\mathsf{ni\_set}(r) := 0 \sqcup \mathsf{ni\_set}(r)$

Figure 22: Abstract interpretation rules to compute ni_set

Proposition 19 (Correctness) *For every term s occurring in a constraint C, if $\mathsf{ni\_set}(s) = i$ then at each step of the computation s is a nested set of integers with a nesting depth equal to i. The analysis terminates* *for every possible input, and its time complexity is $O(nm)$, where n is the number of variables in C and m is the sum of the occurrences of symbols of the terms in C.*

Proof: Correctness follows by structural induction on constraints and terms. The termination is guaranteed by the finite height of the lattice. At every goal scan ($O(m)$ time), at least one variable is updated and raised up by one level in the lattice, otherwise a fixpoint is reached. In the worst case, at each scan only one variable is updated. Since $\mathcal{L}$ is a flat lattice with height equal to 2, after $2n$ scans all the variables will reach the $\top$ value and a fixpoint will be found. Thus, the worst case time complexity is $O(nm)$. □

5.2 Analysis of CLP($\mathcal{SET}, \mathcal{FD}$) Programs

The set of rules presented in the previous section can be used to analyze a complete CLP($\mathcal{SET}, \mathcal{FD}$) program. Without loss of generality, let us assume that each program P is composed of a set of clauses of the form

$$p(X_1, \ldots, X_m) :- C, B$$

where $X_1, \ldots, X_m$ are distinct variables, C is a $\mathcal{SET}\mathcal{FD}$-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. Moreover, we can assume that each variable in the program P occurs only in one clause, with the only exception of the variables in the head of clauses. Given a program P, we assume that all clauses defining the same predicate p use the same set of variables $X_1, \ldots, X_m$—which can be easily accomplished via renaming. Thus, the general form of a procedure is:

$$\begin{array}{lcl} p_1(X_1, \ldots, X_m) & :- & C_1, B_1. \\ & \ldots & \\ p_1(X_1, \ldots, X_m) & :- & C_n, B_n. \end{array}$$

The analysis can gather more information when considering also the program input. Some knowledge about the typical *"types"* of terms expected in a goal can infer refinements and/or detections of some variables' properties. In this context, we will consider the following possible types for an argument:

- *integer*—denoting the fact that the particular argument will always contain an integer
- *set_of_integers*(n)—denoting the fact that the particular argument will always contain a nested set of integers of depth n
- *any*—any other type of value could be present
- *unknown*—no knowledge about this argument is available.

The type of a term s is devised by the static analysis with the following correspondences: $\mathsf{ni\_set}(s) = 0$ implies s is *integer*, $\mathsf{ni\_set}(s) = n > 0$ implies s is a *set_of_integers*(n), $\mathsf{ni\_set}(s) = \top$ implies s is any type and $\mathsf{ni\_set}(s) = \bot$ is *unknown* type.

Example 20 *Let us consider a predicate used to determine the cliques of K elements of an undirected graph $\langle V, E\rangle$, where V is a set of integers (representing vertices of the graph) and E is a set of edges (each represented as a set of two vertices).*

```
clique(V, E, Cliques, K) :-
        ...
```

In this case, the types for the arguments of `clique` *are:*

```
clique(set_of_integers(1), set_of_integers(2), set_of_integers(1), integer)
```

□

Let us discuss the rule to apply for including the type information in the analysis of a program. Let us assume that the goal G contains m variables $X_1, \ldots, X_m$, and that the type information ($type(X_i)$) for these variables is known in advance. Let us add a new clause of the form

$$g(X_1, \ldots, X_m) \;\; :- \;\; X_1 = \alpha(X_1) \,\wedge\, \ldots \,\wedge\, X_n = \alpha(X_n),\; G.$$

to the program P, where α is defined as follows:

(1)	$\emptyset \mapsto$ $\mathtt{dom}(\emptyset) := \bot$
(2)	$t \in Z \mapsto$ $\mathtt{dom}(t) := [t, t]$
(3)	$\left.\begin{array}{r} f(t_1, \dots, t_k) \\ f \in F_Z \end{array}\right\} \mapsto$ $\mathtt{dom}(f(t_1, \dots, t_k)) := f^{\mathcal{D}}(\mathtt{dom}(t_1), \dots, \mathtt{dom}(t_k))$
(4)	$\mathsf{ni\_set}(t) > 1 \mapsto$ $\mathtt{dom}(t) := \top$
(5)	$\mathsf{int}(t_i, t_f) \mapsto$ $\mathtt{dom}(\mathsf{int}(t_i, t_f)) := \mathtt{dom}(t_i) \sqcup \mathtt{dom}(t_f)$
(6)	$\{t \mid s\} \mapsto$ $\mathtt{dom}(\{t \mid s\}) := \mathtt{dom}(t) \sqcup \mathtt{dom}(s);$
(7)	$s = t$ or $s \in t \mapsto$ $\mathtt{dom}(s) := \mathtt{dom}(s) \sqcup \mathtt{dom}(t);$ $\mathtt{dom}(t) := \mathtt{dom}(s)$
(8)	$\cup_3(r, s, t) \mapsto$ $\mathtt{dom}(r) := \mathtt{dom}(r) \sqcup \mathtt{dom}(s) \sqcup \mathtt{dom}(t);$ $\mathtt{dom}(s) := \mathtt{dom}(r);$ $\mathtt{dom}(t) := \mathtt{dom}(r)$

Figure 23: Abstract interpretation rules to compute `dom`

- $\alpha(X) = X$ if $type(X) = unknown$;
- $\alpha(X) = 0$ if $type(X) = integer$;
- $\alpha(X) = \underbrace{\{\dots\{}_{n} 0 \underbrace{\}\dots\}}_{n}$ if $type(X) = set\_of\_integers(n)$;
- $\alpha(X) = a$ if $type(X) = any$, where a is a constant in F_U.

The analysis algorithm proceeds as follows. At the beginning, all program variables are initialized with the value $\bot$ of the lattice $\mathcal{L}$. The program analysis is run until a fixpoint is reached for the $\mathsf{ni\_set}$ values associated to the variables. This is accomplished by repeatedly executing the following two steps for each clause $p(X_1, \dots, X_k) :- C, B$ in the program (the order is irrelevant):

- *Local analysis:* The analysis described in the previous section is applied to C;
- *Propagation of information to other clauses:* For each $q(A_1, \dots, A_k)$ in B, the information contained in the formal parameters $X_1, \dots, X_k$ of q and the actual parameters $A_1, \dots, A_k$ are updated as follows:

$$\begin{array}{rcl} \mathsf{ni\_set}(X_i) & := & \mathsf{ni\_set}(X_i) \sqcup \mathsf{ni\_set}(A_i); \\ \mathsf{ni\_set}(A_i) & := & \mathsf{ni\_set}(X_i); \end{array}$$

5.3 Domain Analysis

The static analysis schema presented can also be used to infer domain information for those variables X such that $\mathsf{ni\_set}(X) \in \{0, 1\}$. For each variable X such that $\mathsf{ni\_set}(X) = 0$—i.e., X is known to be assigned an integer—we define a set, called $\mathtt{dom}(X)$, which contains a superset of possible values that can be assigned to it. In the case $\mathsf{ni\_set}(X) = 1$, the variable X will be assigned a set of integers, and $\mathtt{dom}(X)$ represents a superset of every possible set that can be unified with X at run time.

Let us describe how we can extend the previous analysis framework to determine $\mathtt{dom}()$ as well. In this context, we will rely on approximating an arbitrary set of integers I with the interval $[\min(I), \max(I)]$. The analysis makes use of another abstraction lattice $\mathcal{D}$, whose points are integer intervals ($[x, y]$, $x, y \in \mathbb{Z}$). The order in the lattice is provided by the relation $[x, y] \leq [x', y']$ iff $[x, y] \subseteq [x', y']$. The join operation $\sqcup$ between two intervals is: $[x, y] \sqcup [x', y'] = [\min(x, x'), \max(y, y')]$. This lattice has a least element $\bot$ represented by the empty interval; we also introduce a greatest element $\top$.

The analysis rules, presented in Figure 23, resemble the ones presented for the $\mathsf{ni\_set}$ analysis. In the Figure, we show how we adapt the inductive abstraction to handle the lattice $\mathcal{D}$.

Given an arithmetic function symbol $f \in F_Z$, we define a corresponding abstract operation $f^{\mathcal{D}}$; this is used in rule (3) of the abstract interpretation. In general, for every mathematical operator, its abstraction computes the smallest integer interval that contains every value obtained from the application of the operator to each combination of values from the operands' domains. The definitions for the symbols $+$ and $*$ are shown in Figure 24.

$+^{\mathcal{D}}$	$\bot$	$[\ell_a, u_a]$	$\top$
$\bot$	$\bot$	$\bot$	$\top$
$[\ell_b, u_b]$	$\bot$	$[\ell_a + \ell_b, u_a + u_b]$	$\top$
$\top$	$\top$	$\top$	$\top$

$*^{\mathcal{D}}$	$\bot$	$[\ell_a, u_a]$	$\top$
$\bot$	$\bot$	$\bot$	$\top$
$[\ell_b, u_b]$	$\bot$	$[\min\{\ell_a\ell_b, \ell_b u_a, \ell_a u_b, u_a u_b\}, \max\{\ell_a\ell_b, \ell_b u_a, \ell_a u_b, u_a u_b\}]$	$\top$
$\top$	$\top$	$\top$	$\top$

Figure 24: Definition of $+^{\mathcal{D}}$ and $*^{\mathcal{D}}$

Example 21 *Consider the following CLP($\mathcal{SET}, \mathcal{FD}$) program.*

```
a(X,Y) :-                p(Z) :-
   X ∈ int(1,3),            Z = 3.
   X ≠ Y,                p(Z) :-
   p(Y).                    Z = 5.
```

Running the program analysis on lattices $\mathcal{L}$ and on lattice $\mathcal{D}$ for every variable X such that $\mathsf{ni\_set}(X) = 0$, *we obtain:*

$$\begin{array}{lll} \mathsf{ni\_set}(X) = 0 & \mathsf{ni\_set}(Y) = 0 & \mathsf{ni\_set}(Z) = 0 \\ X \in \mathsf{int}(1,3) & Y \in \mathsf{int}(3,5) & Z \in \mathsf{int}(3,5) \end{array}$$

□

6 CLP($\mathcal{SET}, \mathcal{FD}$) Examples and Experimental Results

In this section we provide some examples and evaluation of the CLP($\mathcal{SET}, \mathcal{FD}$) framework. The CLP($\mathcal{SET}, \mathcal{FD}$) framework has been implemented and it is currently distributed as part of the {log} system.[4] In the specific implementation described here we make use of the constraint solvers offered by SICStus Prolog [27], though the proposed framework is general and applicable to other CLP($\mathcal{FD}$) solvers.

6.1 Sample CLP($\mathcal{SET}, \mathcal{FD}$) Programs

As an example of how constraints on integer terms can be conveniently exploited in CLP($\mathcal{SET}, \mathcal{FD}$) consider the problem of finding solutions for a system of linear equations. The program in Figure 25 shows a CLP($\mathcal{SET}, \mathcal{FD}$) implementation for this problem, while Table 1 shows experimental results obtained running the code using $SAT_{\mathcal{SET}}$ and using $SAT_{\mathcal{SET}+\mathcal{FD}}$. Note that the program code is exactly the same in the two cases, but constraints of the form $X \in \mathsf{int}(a,b)$ in $SAT_{\mathcal{SET}}$ are simply dealt with as usual set membership constraints (namely, $X \in \{a, a+1, a+2, \ldots, b\}$), thus causing the explicit enumeration of all possible bindings for X.

Solving the problem using CLP($\mathcal{SET}$) is feasible but with unacceptable computation time (we indicated with ∞ executions that did not terminate within 1hr). Conversely, the solution using the CLP($\mathcal{SET}, \mathcal{FD}$) solver is obtained in time comparable to that obtained using directly CLP($\mathcal{FD}$) (in which $X \in \mathsf{int}(L,H)$ is replaced by $X \;\mathtt{in}\, L..H$).

[4]The {log} system can be downloaded at `www.math.unipr.it/~gianfr/setlog.Home.html`.

```
start(X,Y,Z,L,H) :-
    X ∈ int(L,H),
    Y ∈ int(L,H),
    Z ∈ int(L,H),
    X1 = 1 + X, X1 = 2 * Y + Z,
    X2 = Z - Y, X2 = 3,
    X3 = X + Y, X3 = 5 + Z.
```

Figure 25: Solution of System of Linear Equations

	Experimental results						
Value of L	-10	-100	-200	-10^3	-10^4	-10^5	-10^6
Value of H	10	100	200	10^3	10^4	10^5	10^6
CLP($\mathcal{SET}$)	0.016s	1.89s	107.4s	∞	∞	∞	∞
CLP($\mathcal{SET},\mathcal{FD}$)	0s	0s	0.015s	0.03s	0.26s	3.4s	40.7s
CLP($\mathcal{FD}$)	0s	0s	0s	0.02s	0.25s	2.842s	33.1s

Table 1: Execution of Program from Figure 25 (Centrino 2GHz, 1GB RAM)

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 code in Figure 26 shows a CLP($\mathcal{SET},\mathcal{FD}$) possible solution for this problem. The $\subseteq$ predicate is a constraint predicate that can be defined using $\cup_3$—i.e., $s \subseteq t \Leftrightarrow \cup_3(s,t,t)$ [12]. $\subseteq$ is used here to constraint each letter of the puzzle to take values in the interval from 0 to 9. The sizeconstraint is used to force the set of variables $\{S,E,N,D,M,O,R,Y\}$ to have cardinality 8, that is to be all different.

```
solve_puzzle(S,E,N,D,M,O,R,Y) :-
        {S,E,N,D,M,O,R,Y} ⊆ int(0,9),
        size({S,E,N,D,M,O,R,Y},8),
        S ≠ 0, M ≠ 0,
        M * 10000 + O * 1000 + N * 100 + E * 10 + Y  =
                  S * 1000 + E * 100 + N * 10 + D  +
                  M * 1000 + O * 100 + R * 10 + E.
```

Figure 26: Send+More=Money Puzzle

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.

Another CLP($\mathcal{SET},\mathcal{FD}$) program is shown in Figure 27. It is a simple example of training a weighted automata using samples and unknown transition weights. The program expects as inputs:

- a directed automata, represented as a set of edges, where each edge is a tuple $(Start, End, Character)$;
- a set of samples, of the form $(String, Cost)$, where $String$ is a list of symbols from the alphabet of the automata, and $Cost$ is an integer representing the desired weight for such string.

The output (`Automata`) is a weighted version of the input automata, where each edge has a cost information (`cost(EdgeCost)`); the computation will restrict the value of each `EdgeCost` to the ranges that guarantee the correct cost for the input samples. Initially, each weight is restricted to the range $[1, 1000]$.

The process of validating each sample consists of creating the collection of edges required to process the input string (Path), extracting the actual edges (with their costs) from the automata, and finally verifying that the resulting cost of the path matches the cost specified for the sample string.

$$
\begin{array}{l}
\texttt{training}(\texttt{E},\texttt{Samples},\texttt{Automata}) :- \\
\quad \texttt{create\_automata}(\texttt{E},\texttt{Automata}), \\
\quad (\forall\,(\texttt{String},\texttt{Cost}) \in \texttt{Samples})(\texttt{validate}(\texttt{Automata},\texttt{String},\texttt{Cost})). \\
\texttt{create\_automata}(\emptyset,\emptyset). \\
\texttt{create\_automata}(\{(\texttt{Start},\texttt{End},\texttt{Char}) \mid \texttt{Rest}\},\{(\texttt{Start},\texttt{End},\texttt{Char},\texttt{cost}(\texttt{X})) \mid \texttt{NewRest}\}) :- \\
\quad \texttt{X} \in \mathsf{int}(1,1000), \\
\quad (\texttt{Start},\texttt{End},\texttt{Char}) \notin \texttt{Rest}, \\
\quad \texttt{create\_automata}(\texttt{Rest},\texttt{NewRest}). \\
\texttt{validate}(\texttt{Automata},\texttt{String},\texttt{Cost}) :- \\
\quad \texttt{make\_path}(\texttt{String},\texttt{Path},\texttt{q}_0,\texttt{Cost}), \\
\quad \texttt{Path} \subseteq \texttt{Automata}. \\
\texttt{make\_path}([\,],\emptyset,\_,0). \\
\texttt{make\_path}([\texttt{Char}\mid\texttt{Rest}],\{(\texttt{State},\texttt{End},\texttt{Char},\texttt{cost}(\texttt{EdgeCost})) \mid \texttt{PathRest}\},\texttt{State},\texttt{EdgeCost}+\texttt{Cost}) :- \\
\quad \texttt{make\_path}(\texttt{Rest},\texttt{PathRest},\texttt{End},\texttt{Cost}).
\end{array}
$$

Figure 27: Training of an Automata

The $\forall$ are used to encode restricted universal quantifiers, as discussed in Section 2.2. This simple form of RUQ can be generalized to the more complex form

$$\mathsf{forall}(t \in s, \exists \bar{Z} \varphi[\bar{Y}, \bar{Z}])$$

where t is a term and $\bar{Y} = vars(t)$.

For instance, assume that we use as input $\texttt{E} = \{(\texttt{q}_0,\texttt{q}_0,1),(\texttt{q}_0,\texttt{q}_1,0),(\texttt{q}_1,\texttt{q}_0,1)\}$. In Figure 28, we represent the corresponding automata, where we associate to each edge the Char and the Cost(). Suppose to input the set of samples $\texttt{Samples} = \{(101,6),(110,7)\}$. The call to the training predicate returns $\texttt{Automata} = \{(\texttt{q}_0,\texttt{q}_0,1,\texttt{cost}(\texttt{A})),(\texttt{q}_0,\texttt{q}_1,0,\texttt{cost}(\texttt{B})),(\texttt{q}_1,\texttt{q}_0,1,\texttt{cost}(\texttt{C}))\}$, where the domains are restricted and enumerated using: $\texttt{A} \in \mathsf{int}(1,3), \texttt{B} \in \mathsf{int}(1,5), \texttt{C} \in \mathsf{int}(0,4)$. Note that the restriction is efficiently operated by interval propagation on the linear equations generated by make_path (i.e., $\texttt{A}+\texttt{B}+\texttt{C} = 6, 2\texttt{A}+\texttt{B} = 7$). Consider a more specified example, in which we add an extra sample: $\texttt{Samples} = \{(101,6),(110,7),(10,5)\}$. In this case the automaton returned contains a fully specified Cost for each edge: $\texttt{A} = 2, \texttt{B} = 3, \texttt{C} = 1$. Using the slightly modified samples set $\texttt{Samples} = \{(101,6),(110,7),(10,3)\}$, produces a failure.

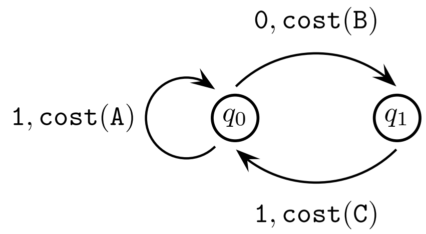

Figure 28: Example of Training of an Automata

As a final example, we show a program dealing with a multiset data structure. The availability of set and integers allows a rather natural encoding of multisets. For instance, a multiset with three occurrences of the token a and four occurrences of the token b can be represented as:

$$\{\mathtt{el}(a,3),\mathtt{el}(b,4)\}$$

It is therefore easy to encode algorithms on multisets and in particular those based on multiset rewriting. For instance, we can encode an interpreter for the GAMMA language [3] in the following way:

```
gamma(Input, Input) :-
    not rule(Input, _).
gamma(Input, Output) :-
    rule(Input, Intermediate),
    gamma(Intermediate, Output).
```

Figure 29: A GAMMA interpreter

GAMMA rewriting rules can be simply defined as rules of the form:

$$\mathtt{rule(InputMultiset, OutputMultiset)} :- \textit{Condition}$$

where `InputMultiset` and `OutputMultiset` are multiset terms and *Condition* is a goal representing the *reaction condition* of the rule, namely the condition that must be satisfied in order to apply the rule. For example, the following rewriting rules govern the chemical reaction composing water molecules:

```
rule({el(h, M), el(o, N)}, {el(h, M − 2), el(o, N − 1), el(water, 1)} :-
    2 ≤ M, 1 ≤ N.
rule({el(h, M), el(o, N), el(water, P)}, {el(h, M − 2), el(o, N − 1), el(water, P + 1)}) :-
    2 ≤ M, 1 ≤ N.
rule({el(A, 0) | R}, R) :-
    el(A, 0) ∉ R.
```

The output to a goal of the form:

$$:-\ \mathtt{gamma}(\{\mathtt{el}(\mathtt{o},6),\mathtt{el}(\mathtt{h},13)\},\mathtt{Output}).$$

will be

$$\mathtt{Output} = \{\mathtt{el}(\mathtt{water},6),\mathtt{el}(\mathtt{h},1)\}.$$

6.2 Application of the Static Analysis Framework

The results of the static analysis can be exploited to improve the constraint solving process in various respects.

A first improvement consists of refining the constraint rewriting rules by allowing them to exploit information provided by the ni_set measure. Specifically, in the current definition of the rules, constraints of the forms of the forms $s \in \mathsf{int}(t_1,t_2)$, $s \in \{t_1,\ldots,t_n\}$, $s = t$, $s \neq t$, where s or t are variables, are always passed to the $\mathcal{FD}$ solver. However, if s or t are subsequently instantiated to non-integer terms, this communication is useless. Note that the constraint solving process is nevertheless correct thanks to the generation of the integer constraints. Using the ni_set information we can refine the rules so that only if the values of $\mathsf{ni\_set}(s)$ and $\mathsf{ni\_set}(t)$ are 0 or $\perp$ the constraint is passed to $SAT_{\mathcal{FD}}$; otherwise, it is completely solved by the $\mathcal{SET}$ solver.

Further improvements can be obtained by exploiting the results of the domain analysis. Availability of domain information allows, in general, to obtain more precise results, possibly in shorter time, from the $\mathcal{FD}$ solver. Domain analysis allows us to introduce in advance new constraints of the form $X \in \mathtt{dom}(X)$ for each variable X such that $\mathsf{ni\_set}(X) \in \{0,1\}$.

Example 22 *Let* n *be an arbitrary natural number. Consider the following program, depending on* n*:*

```
p(X,Y) :-            q_1(X) :-                r_1(Y) :-
   X ≠ Y,               X ≠ n + 1 ∧ q_2(X).      Y ≠ 1 ∧ r_2(Y).
   q_1(X),           q_2(X) :-                r_2(Y) :-
   r_1(Y).              X ≠ n + 2 ∧ q_3(X).      Y ≠ 2 ∧ r_3(Y).
                     ⋮                        ⋮
                     q_n(X) :-                r_n(Y) :-
                        X ≠ 2n ∧                 Y ≠ n ∧
                        X ∈ int(1,n).            Y ∈ int(n+1,2n).
```

Static analysis allows us to detect immediately that the domains of X *and* Y *are disjoint, and thus it is immediately detected that* X ≠ Y *is true, and thus* :− p(X,Y) *holds.* □

```
schur(N,S):-              subsetint(S,M,N):-
  subsetint(S,1,N),         X in M .. N,
  s1(S,S),                  lengthfd(S,X),
  labeling([],S).           domain(S,M,N),
                            orderedfd(S).

s1([],S).
s1([X|R],S):-             lengthfd([],0).
  nonmemberfd(X,R),       lengthfd([_|R],N):-
  s2(S,S,X),                N #> 0,
  s1(R,S).                  M #= N - 1,
                            lengthfd(R,M).

s2([],S,X).
s2([Y|R],S,X):-           orderedfd([]).
  nonmemberfd(Y,R),       orderedfd([_]).
  s3(S,S,X,Y),            orderedfd([A,B|R]):-
  s2(R,S,X).                A #< B,
                            orderedfd([B|R]).

s3([],S,X,Y).
s3([Z|R],S,X,Y):-         nonmemberfd(A,[]).
  nonmemberfd(Z,R),       nonmemberfd(A,[X|R]):-
  Z #\= X+Y,                 A #\= X,
  s3(R,S,X,Y).               nonmemberfd(A,R).
```

Figure 30: The compiled program

As a further possible improvement, static analysis can be used, in some cases, to directly compile a fragment of a CLP($\mathcal{SET}, \mathcal{FD}$) program P into CLP($\mathcal{FD}$). This can be done, for example, when, for all terms in P we have that $\mathsf{ni\_set}(s) \in \{0,1\}$. For example, 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 (this problem is a simplification of the problem of computing Schur numbers):

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

The static analysis process is applied to the program obtained from the removal of the RUQs. The static analysis produces

$$\mathsf{ni\_set}(N) = \mathsf{ni\_set}(X) = \mathsf{ni\_set}(Z) = \mathsf{ni\_set}(Y) = 0$$
$$\mathsf{ni\_set}(S) = 1$$

These results can be used to directly derive the SICStus CLP($\mathcal{FD}$) program in Figure 30. The predicate `subsetint` is a standard translation of the built-in $\subseteq$ when its second argument is an interval. The last three predicates on the right are auxiliary predicates that do not depend on the specific program. In Table 2 we report the running times obtained by running the same program in CLP($\mathcal{SET}$) and CLP($\mathcal{SET},\mathcal{FD}$) with the given automatic transformation to CLP($\mathcal{FD}$). The programs have been executed for different values of N, using a Laptop Centrino 2GHz with 1GB RAM, using SICStus Prolog 3.12.2. Precisely, in CLP($\mathcal{SET}$) the goal is `Q={A:schur(N,A)}` and in SICStus (execution of the program automatically derived from CLP($\mathcal{SET},\mathcal{FD}$) using static analysis) the goal is `setof(A,schur(N,A),Q)`.

N	5	6	7	8	12	16
CLP($\mathcal{SET}$)	0.594s	2.625s	11.53s	51.76s	∞	∞
CLP($\mathcal{SET},\mathcal{FD}$)	0.016s	0.031s	0.062s	0.125s	2.61s	46.3s
CLP($\mathcal{FD}$)	0.000s	0.000s	0.000s	0.016s	0.17s	1.23s

Table 2: Computational results for the `schur` program (Centrino 2GHz, 1GB RAM)

7 Conclusions

In this work we presented a novel constraint logic programming framework, CLP($\mathcal{SET},\mathcal{FD}$), which combines algorithms for high-level symbolic manipulation of sets and domains with the efficient manipulation of constraints on finite domains. On one hand, the framework extends the possibilities of existing finite domain solvers by providing the ability to view and manipulate domains as first-class entities. On the other hand, the framework improves efficiency of existing proposals on constraint solving with hereditarily finite sets by automatically discovering instances of finite domain constraints and mapping them to fast, propagation-based solvers.

The CLP($\mathcal{SET},\mathcal{FD}$) framework defines the syntax and the semantics of the extended constraint logic programming language, along with sound and complete procedures to handle all the admissible constraints. In order to improve efficiency and flexibility, the framework includes also an abstract interpreter aimed at automatically detecting instances of set constraints that can be reduced to finite domain constraints.

A number of extensions of the framework are currently under consideration. First of all, most existing $\mathcal{FD}$ solvers provide more extensive coverage of constraints (e.g., global constraints), which might further simplify the handling of CLP($\mathcal{SET},\mathcal{FD}$) constraints. We propose to extend the abstract analysis framework to strengthen the capability of converting entire program fragments to $\mathcal{FD}$ programs. We are also investigating ways to tie the solvers at a lower level—i.e., at the level of indexicals—to improve the impact of propagation. Another direction is that of developing special constraints solving procedures for sets of FD elements (as done in [15]) and for sets of sets of FD elements. Graphs are special cases of sets of sets of FD elements and constraints on them have been recently studied in [11].

References

[1] Abrial, J-R. (1996) *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Arenas-Sánchez, P., and Rodríguez-Artalejo, M. (2001) A General Framework for Lazy Functional Logic, Programming with Algebraic Polymorphic Types. *Theory and Practice of Logic Programming*, 2(1):185–245.

[3] BÂNATRE, J., AND LE MÉTAYER, D. (1993) Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111.

[4] BELDICEANU, N., AND CARLSSON, M. (2001) Revisiting the Cardinality Operator and Introducing the Cardinality-Path Constraint Family. In *International Conference on Logic Programming*, P. Codognet, Ed., Springer Verlag, pp. 59–73.

[5] 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.

[6] CARLSSON, M., OTTOSSON, G., CARLSON, B. (1997) An Open-Ended Finite Domain Constraint Solver. In H. Glaser, P. H. Hartel, and H. Kuchen eds. *Principles of Declarative Programming.* Vol. 1292 of LNCS, Springer Verlag, pp. 191–206.

[7] CODOGNET, P., AND DIAZ, D. (1996) Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3):185–226.

[8] 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, pp. 219–229.

[9] DILLER, A. (1994) *Z: An Introduction to Formal Methods.* J. Wiley & Sons.

[10] DOVIER, A., OMODEO, E. G., PONTELLI, E., AND ROSSI, G. (1996) {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44.

[11] DOOMS, G., DEVILLE, Y., DUPONT, P. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. Proc. of CP-2005, pp. 211–225, LNCS 3709, 2005.

[12] 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.

[13] FLENER, P., HNICH, B., AND KIZILTAN, Z. (2001) Compiling high-level type constructor in constraint programming. In I.V. Ramakrishnan ed., *Practical Aspects of Declarative Languages*, Vol. 1990 of LNCS, Springer Verlag, pp. 229–244.

[14] GAVANELLI M., LAMMA E., MELLO P., AND MILANO, M. (2005) Dealing with incomplete knowledge on CLP(FD) variable domains, In *ACM Transactions on Programming Languages and Systems*, 27(2):236–263.

[15] GERVET, C. (1997) Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244.

[16] HOFSTEDT, P. (2000) Cooperating Constraint Solvers. R. Dechter Ed., *International Conference on Principle and Practice of Constraint Programming*, Vol. 1894 of LNCS, Springer Verlag, pp. 520–524.

[17] IC PARC (2003) *The ECLiPSe Constraint Logic Programming System.* London. `www.icparc.ic.ac.uk/eclipse/`.

[18] JAFFAR, J., AND MAHER, M. J. (1994) Constraint Logic Programming: A Survey. *Journal of Logic Programming* 19–20:503–581.

[19] JAFFAR, J., MAHER, M.J., MARRIOTT, K., AND STUCKEY, P.J. (1998) The Semantics of Constraint Logic Programs. *Journal of Logic Programming*, 37(1–3):1–46.

[20] JAYARAMAN, B. (1992) Implementation of Subset-Equational Programs. *Journal of Logic Programming*, 12(4):299–324.

[21] Lassez, J.-L., Maher, M.J., and Marriot, K. (1988) Unification Revisited. *Foundation of Deductive Deatabases and and Logic Programming*, Morgan Kaufmann, pp. 587–625.

[22] Marriott, K., and Stuckey, P. J. (1998) *Programming with Constraints: an Introduction.* The MIT Press, Cambridge, Mass.

[23] Quimper, G.-C., and Walsh, T. Beyond Finite Domains: the All Different and Global Cardinality Constraints. Proc. of CP-2005, pp. 812–816, LNCS 3709, 2005.

[24] Régin, J.-C. (1996) Generalized Arc Consistency for Global Cardinality Constraints. In *Proceedings of the AAAI/IAAI Conference* AAAI/MIT Press, pp. 209–215.

[25] Rossi, G. (2005) *The {log} Constraint Logic Programming Language.* `prmat.math.unipr.it/~gianfr/setlog.Home.html`.

[26] Shmueli, O., Tsur, S., and Zaniolo, C. (1992) Compilation of Set Terms in the Logic Data Language (LDL). *Journal of Logic Programming*, 12(1–2):89–119.

[27] Swedish Institute of Computer Science. *The SICStus Prolog Home Page.* `www.sics.se`.

[28] Van Hentenryck, P. (1989) *Constraint Satisfaction in Logic Programming.* MIT Press.

[29] Van Hentenryck, P., and Deville, Y. (1991) The Cardinality Operator: a New Logical Connective for Constraint Logic Programming. In K. Furukawa, Ed., *International Conference on Logic Programming*, MIT Press, pp. 745–759.

[30] Wang, L., Wijesekera, D., and Jajodia, S. (2004) A Logic-based Framework for Attribute based Access Control. V. Athuri et al., Eds., *2nd ACM Workshop on Formal Methods in Security Engineering*, ACM Press, pp. 45–55.

[31] J. Wielemaker. (2004) *SWI-Prolog Reference Manual (Versin 5.4).* University of Amsterdam.

[32] Yakhno, T., and Petrov, E. (2000) Extensional Set Library for ECLiPSe. In *Perspectives of System Informatics*, Vol. 1755 of LNCS, Springer Verlag, pp. 434–444.

[33] Zhou, N-F. (2005) *B-Prolog User's Manual (Version 6.8).* Afany Software.