Extending the $\mathcal{TOY}$ System with the ECL^{i}PS^e Solver over Sets of Integers

S. Estévez-Martín[1], J. Correas[1] and F. Sáenz-Pérez[2] *

[1]DSIC, [2]DISIA
Complutense University of Madrid
Madrid, Spain

Abstract. Starting from a computational model for the cooperation of constraint domains in the CFLP context (with lazy evaluation and higher-order functions), we present the theoretical basis for the coordination domain $\mathcal{C}$ tailored to the cooperation of three pure domains: the domain of finite sets of integers ($\mathcal{FS}$), the finite domain of integers ($\mathcal{FD}$) and the Herbrand domain ($\mathcal{H}$). We also present the adaptation of the goal-solving calculus $CCLNC(\mathcal{C})$ (Cooperative Constraint Lazy Narrowing Calculus over $\mathcal{C}$) to this particular case, as well as soundness and limited completeness results. An implementation of this cooperation in the CFLP system $\mathcal{TOY}$ is presented. Our implementation is based on inter-process communication between $\mathcal{TOY}$ and the external solvers for sets of integers and finite domain of ECL^{i}PS^e.
Keywords: Constraint Functional Logic Programming, Constrained Lazy Narrowing, Implementation, Domain Cooperation.

1 Introduction

$\mathcal{TOY}$ [14] is a multiparadigm programming language and system designed to integrate the main declarative programming styles and their combination. $\mathcal{TOY}$ supports constraint functional logic programming, including symbolic equations and disequations, arithmetic constraints over the real numbers, and finite domain constraints. $\mathcal{TOY}$ has also incorporated a mechanism to support solver cooperation, and a detailed description of both the theoretical foundations and practical implementations involving three specific domains (Herbrand $\mathcal{H}$, reals $\mathcal{R}$ and finite domains of integers $\mathcal{FD}$) is given in [5]. This model relies on the $CFLP(\mathcal{D})$ scheme [11] and the goal solving calculus [10]. This scheme must be instantiated to a concrete constraint domain $\mathcal{D}$ which provides specific data values, constraints based on specific primitive operations, and a constraint solver. In particular, this concrete constraint domain is the so-called *Coordination Domain* $\mathcal{C} = \mathcal{M} \oplus \mathcal{D}_1 \oplus \cdots \oplus \mathcal{D}_n$, and is constructed from *pure* domains (i.e., each $\mathcal{D}_i$), given as components, and a *Mediatorial Domain* $\mathcal{M}$. This domain supplies

* This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01) (first and third authors), DOVES (TIN-2008-05624) (second author), Prometidos-CM (S2009TIC-1465), and GPD (UCM-BSCH-GR35/10-A-910502).

communication mechanisms among these pure domains using a special kind of *hybrid* constraints called *bridges* which allow to exchange information between pure domains.

Finite domain and set solvers have been widely studied in CLP, e.g. ECL^{i}PS^e, Conjunto and Cardinal [7, 2, 1], and also in functional programming (FaCiLe). However, to the best of our knowledge, this cooperation has not been considered in the context of CFLP languages. Curry and its prolog-based implementation, PAKCS, do not incorporate a solver for set constraints.

This paper, based on [5], contains the following contributions: First, a computational model for the cooperation of the Herbrand domain $\mathcal{H}$, the finite domain $\mathcal{FD}$, and a finite set domain $\mathcal{FS}$ is presented. The finite set domain has been designed as an improved extension of ECL^{i}PS^e set solver, as well as a mediatorial domain $\mathcal{M}$ suitable for this cooperation. The computational model has been proved sound and complete, with some restrictions w.r.t. the declarative semantics provided by the $CFLP(\mathcal{D})$ scheme. And finally, a preliminary implementation that integrates ECL^{i}PS^e set solver into $\mathcal{TOY}$ has been developed.

Regarding the implementation, in a previous paper [4] we presented as a first approach a solver for sets of integers adapting a generic framework for defining and solving interval constraints on any set of domains (finite or infinite) that are lattices [6]. That experimental set solver prototype was not efficient enough. Moreover, a formal computational model was not provided. In this paper, we include a different implementation that uses set libraries provided by the constraint logic programming system ECL^{i}PS^e [12, 13, 8] .

This paper is organized as follows: Section 2 introduces the language $\mathcal{TOY}$. Section 3 defines the domains $\mathcal{FS}$ and $\mathcal{M}$. In Section 4 we extend the formal goal-solving calculus called $CCLNC(\mathcal{C})$ (Cooperative Constraint Lazy Narrowing Calculus over $\mathcal{C}$) with new rules explicitly defined for this cooperation. Section 5 presents a prototype of this cooperation in the $\mathcal{TOY}$ system, and benchmarks taken straight from the ECL^{i}PS^e distribution site [3] are used to compare with our approach in terms of performance with encouraging results. Finally, Section 6 concludes and presents lines for future work.

2 The $\mathcal{TOY}$ Language

$\mathcal{TOY}$ [14] is a functional logic language and system that solves goals by means of a *demand driven lazy narrowing strategy* [9] combined with constraint solving.

A $\mathcal{TOY}$ program is a collection of definitions of datatypes, operators, lazy defined functions in Haskell style, as well as definitions of predicates in Prolog style. A *predicate* is a particular kind of function which returns a Boolean true value. A *function* includes an optional *polymorphic type declaration* $f :: \tau_1$ `->` ... τ_n `->` τ, and one or more defining rules with *curried notation*:

$$f\ t_1\ \ldots\ t_n = r\ \texttt{<==}\ C_1, \ldots, C_m\ \texttt{where}\ D_1, \ldots, D_p$$

A defined function must be *linear* in the *left-hand side*, i.e., $t_1\ \ldots\ t_n$ cannot include repeated variables. The *right-hand side* r can be any expression built

from variables, built-in operations, data constructors, and functions. *Conditions* C_i, and *local definitions* D_j are optional. The intended meaning of a defining rule is the same as in functional programming languages, namely: an expression of the form $f\ e_1 \ldots e_n = r$ can be reduced by reducing the actual parameters e_k until they match the patterns t_i, checking that the conditions C_i are satisfied, and then reducing the right hand side r (affected by the pattern matching substitution); all this by using the local definitions D_j to obtain the values of locally defined variables. Functions can be *non-deterministic* and *higher-order*. A *goal* in $\mathcal{TOY}$ is a conjunction of conditions $C_1, \ldots, C_m$ where each condition C_i is interpreted as a constraint to be solved.

Example 1. To illustrate the $\mathcal{TOY}$ language, let us define the following function `atMostOne` which, given a list of sets, restricts it to a list of triples such that, given any two sets in the list, they have at most one element in common. The symbol `#--` stands for cardinality.

```
atMostOne :: [set]->bool
atMostOne [] = true
atMostOne [X|Xs] = atMostOne Xs <==
    doList X Xs

doList :: set->[set]->bool
doList _ [] = true
doList X Ys = andL (map (sendCon X) Ys)
```

```
sendCon :: set->set->bool
sendCon SX SY = true <==
    3 #-- SX, 3 #-- SY,
    intersect SX SY SZ,
    Z #-- SZ, Z #<= 1

andL :: [bool]->bool
andL = foldr (/\) true
```

The function `doList` uses the higher-order library function `map`, which applies a partial application `sendCon X` to each element of the list `Ys`, returning a list of Boolean results. A possible goal is:

`L==[S1,S2]`, `domainSets L {} {1,2,3,4,5}`, `atMostOne L`, `labelingSets L`

where `domainSets L {} {1,2,3,4,5}` constrains each set variable, `S1` and `S2`, to be lattices of sets between {} and {`1,2,3,4,5`}; `labelingSets` enumerates all ground values of the variables `S1` and `S2`. Some solutions of this goal are:

`S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {1,4,5} `S1` $\mapsto$ {2,3,5}, `S2` $\mapsto$ {1,2,4}
`S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {2,4,5} `S1` $\mapsto$ {2,3,5}, `S2` $\mapsto$ {1,3,4}

Although in this paper curly brackets are used for denoting sets, our current implementation resorts to the data constructor `s`, as such brackets are syntactically reserved for delimiting sections. □

3 Constraint Domains and Solvers

In this section we present the formal definition of two constraint domains and their solvers, the pure domain $\mathcal{FS}$ and the mediatorial domain $\mathcal{M}$. Both are defined with respect to the computational model for the cooperation of constraint domains described in [5]. First let us recall some aspects.

A constraint domain has a specific signature $\Sigma = \langle TC, SBT, DC, DF, SPF \rangle$ consisting of pairwise disjoint sets of *Type Constructors* (TC), *Specific Base Types* (SBT), *Data Constructors* (DC), *Defined Function Symbols* (DF) and

Specific Primitive Function Symbols (SPF). Base types and primitive function symbols are related to specific constraint domains, while type constructors, data constructors, and defined function symbols are provided by user programs.

In our computational model a *constraint domain* $\mathcal{D}$ with signature Σ is a structure $\mathcal{D} = \langle \mathcal{B}^{\mathcal{D}}, \{p^{\mathcal{D}}\}_{p \in SPF} \rangle$ where $\mathcal{B}^{\mathcal{D}} = \bigcup_{d \in SBT} \mathcal{B}^{\mathcal{D}}_d$ is the set of *base values* and $p^{\mathcal{D}}$ is the *interpretation* of each primitive function symbol $p \in SPF$. Interpretation $p^{\mathcal{D}} \overline{t}_n \to t$ means that the primitive function p with given arguments $t_1, \ldots, t_n$ returns t in the domain $\mathcal{D}$.

We define an *expression* with the syntax $e ::= X \mid \bot \mid u \mid h \mid (e\, e_1)$, where X is a variable, $\bot$ is a special data constructor that represents an undefined value that belongs to any type, $u \in \mathcal{B}^{\mathcal{D}}$ is a base value, $h \in DC \cup DF \cup SPF$, and $(e\, e_1)$ stands for the application of e to e_1. A *pattern* is a particular expression representing a data value that does not need to be evaluated. Its syntax is $t ::= X \mid \bot \mid u \mid c\,\overline{t}_m \mid f\,\overline{t}_m \mid p\,\overline{t}_m$, where X is a variable, $u \in \mathcal{B}^{\mathcal{D}}$, $c \in DC^n$ for some $m \leq n$, $f \in DF^n$ for some $m < n$, and $p \in SPF^n$ for some $m < n$, representing partial applications. Expressions and patterns without any occurrence of $\bot$ are called total.

An *atomic constraint* $\pi \in ACon_{\mathcal{D}}$ over a given domain $\mathcal{D}$ is defined either as $\Diamond$ (standing for *truth*), or $\blacklozenge$ (standing for *falsity*), or $p\ \overline{e}_n \to\!!\ t$ with $p \in SPF^n$, where each e_i is an expression and t is a total pattern. By convention, constraints of the form $p\ \overline{e}_n \to\!!\ \mathit{true}$ are abbreviated as $p\ \overline{e}_n$. In particular, strict equality constraints e_1 `==` e_2 and strict disequality constraints e_1 `/=` e_2 are understood as abbreviations of `(==)` $e_1\ e_2 \to\!!$ `true` and `(==)` $e_1\ e_2 \to\!!$ `false`, respectively. *Atomic primitive constraints* $\pi \in APCon_{\mathcal{D}}$ over a given domain $\mathcal{D}$ are atomic constraints where $\overline{e}_n$ are patterns.

A *substitution* $\sigma \in Sub_{\mathcal{D}}$ over a given domain $\mathcal{D}$ is a set of mappings from variables to patterns. A *valuation* η over a given domain $\eta \in Val_{\mathcal{D}}$, is a ground substitution, that maps variables to values. The valuations that satisfy a given constraint π are said to be *solutions of* π, $Sol_{\mathcal{D}}(\pi) \subseteq Val_{\mathcal{D}}$.

A *constraint store* is a pair $S = \Pi \Box \sigma$ for a domain $\mathcal{D}$, where $\Pi \subset APCon_{\mathcal{D}}$ is a set of atomic primitive constraints and σ is an idempotent substitution such that domain variables of σ and variables of Π are disjoint. The symbol $\Box$ is interpreted as a conjunction. *Solutions of constraint stores* are defined as $Sol_{\mathcal{D}}(\Pi \,\Box\, \sigma) = Sol_{\mathcal{D}}(\Pi) \cap Sol_{\mathcal{D}}(\sigma)$ where $Sol_{\mathcal{D}}(\sigma) = \{\eta \in Val_{\mathcal{D}} \mid \eta = \sigma\eta\}$. If σ is empty $Sol_{\mathcal{D}}(\Pi)$ is used.

Constraint domains are equipped with their respective solvers, which process the constraints arising in the course of a computation. We consider a constraint solver for the domain $\mathcal{D}$ as modeled by a function $solve^{\mathcal{D}}$. From a user viewpoint, a solver can behave as a black-box or a glass-box. We use a convenient abstract technique for specifying the behaviour of glass-box solvers named *store transformation system*. The idea is to specify a set of *store transformation rules* which describe different ways to transform a given store. A store is called *irreducible* iff no store transformation rules can be applied to transform it. A rule is not applicable if the store is not transformed by the rule in any way.

A *transformation step* for a given rule is denoted as $\vdash_{\mathcal{D}}$ and

– $\pi, \Pi \,\square\, \sigma \vdash_{\mathcal{D}} \Pi' \,\square\, \sigma'$ indicates that the store $\pi, \Pi \,\square\, \sigma$, which includes the atomic constraint π plus other constraints Π, is transformed into $\Pi' \square \sigma'$ in one step.
– $\Pi \,\square\, \sigma \vdash^*_{\mathcal{D}} \Pi' \,\square\, \sigma'$ indicates that $\Pi \,\square\, \sigma$ can be transformed into $\Pi' \,\square\, \sigma'$ in finitely many steps.
– $\Pi \,\square\, \sigma \vdash_{\mathcal{D}} \blacksquare$ indicates a failing transformation step.

Solvers reduce primitive constraints to *solved forms*, which are simpler and are shown as computed answers to the users. Formally:

$SF_{\mathcal{D}}(\Pi \square\ \sigma) = \{\Pi' \ \square\ \sigma' \mid \Pi \ \square\ \sigma \vdash^*_{\mathcal{D}} \Pi' \ \square\ \sigma',$ and $\Pi' \ \square\ \sigma'$ is irreducible$\}$

3.1 The Constraint Domain $\mathcal{FS}$ (Finite Sets)

We define the $\mathcal{FS}$ domain with the domain specific signature $\langle TC,\ SBT_{\mathcal{FS}},\ DC,\ DF,\ SPF_{\mathcal{FS}}\rangle$ where $SBT_{\mathcal{FS}} = \{\texttt{elem}, \texttt{set}\}$. The set of base values of base type `elem` is a denumerable set $\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}}$ with a strict total order $\prec$, and the set of base values for `set` contains all finite subsets of $\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}}$, $\mathcal{B}^{\mathcal{FS}}_{\texttt{set}} = \mathcal{P}_f(\mathcal{B}^{\mathcal{FS}}_{\texttt{elem}})$. A set of elements of type `elem` is represented as $\{e_1, \ldots, e_n\}$ where $e_1 \prec \ldots \prec e_n$. Primitive functions in $SPF_{\mathcal{FS}}$ are the following:

– Strict equality and strict disequality `(==), (/=) :: A -> A -> bool`.
– `domainSets :: [set]->set->set->bool` constrains the domain of each set variable of the list w.r.t. a given lower and upper set bounds forming a lattice of possible values for each variable.
– `subset`, `superset :: set->set->bool`;
`intersect`, `union :: set->set->set` are the usual operations on sets.
– `disjoints :: [set]->bool` constrains the list of sets to have no element in common.
– `labelingSets :: [set]->bool` enumerates all ground instantiations of each set expression in the list.
– $\prec$ `:: elem->elem->bool` is the strict total order between set elements.
– `isin :: elem->set->bool` constrains an element to be a member of a set.

Example 2. In order to illustrate the behaviour of some primitive functions, the $\mathcal{TOY}$ goal: `domainSets [Sx,Sy,Sz] {} {1,2}`, `subset Sx Sy`, `subset Sy Sz`, `disjoints [Sx,Sz]`, `superset Sy {1}`, `labelingSets [Sx,Sy,Sz]` produces three solutions: `Sx` $\mapsto$ {}, `Sy` $\mapsto$ `{1,2}`, `Sz` $\mapsto$ `{1,2}`; `Sx` $\mapsto$ {}, `Sy` $\mapsto$ `{1}`, `Sz` $\mapsto$ `{1,2}`; and `Sx` $\mapsto$ {}, `Sy` $\mapsto$ `{1}`, `Sz` $\mapsto$ `{1}`. □

3.2 Solver for $\mathcal{FS}$ Domain

The $\mathcal{FS}$ solver for $\mathcal{TOY}$ has been developed on top of the solver for sets of integers available in ECL^{i}PS^e, and has been extended with several additional features, such as disequality handling and constraint deduction. For this reason, $\mathcal{FS}$ solver has been split in two different layers: First, a glass-box solver $\mathcal{FS}^{\mathcal{T}}$ in $\mathcal{TOY}$ for dealing with those additional features, and second, the black-box solver $\mathcal{FS}^{E}$ available in ECL^{i}PS^e. Observe that, in the formal description of the

S1	`domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb /=` $\{\}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `domainSets` $[S_1, \ldots, S_n]$ `lb ub`, `lb /=` $\{\}$, S_1 `/=` $\{\}$,$\ldots$, S_n `/=` $\{\}$, $\Pi \Box \sigma$
S2	`isIn` X S, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `isIn` X S, S `/=` $\{\}$, $\Pi \Box \sigma$
S3	`subset` S_1 S_2, S_1 `/=` $\{\}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `subset` S_1 S_2, S_1 `/=` $\{\}$, S_2 `/=` $\{\}$, $\Pi \Box \sigma$
S4	`subset` S_1 S_2, S_2 `==` $\{\}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_2 `==` $\{\}$, S_1 `==` $\{\}$, $\Pi \Box \sigma$
S5	`intersect` S_1 S_2 S_3, S_3`/=`$\{\}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `intersect` S_1 S_2 S_3, S_3`/=`$\{\}$,S_1 `/=` $\{\}$, S_2 `/=` $\{\}$, $\Pi \Box \sigma$
S6	`union` S_1 S_2 S_1, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_1, `subset` S_2 S_1, $\Pi \Box \sigma$
S7	`union` S_1 S_2 S_2, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `union` S_1 S_2 S_2, `subset` S_1 S_2, $\Pi \Box \sigma$
S8	`union` S_1 S_2 S_3, S_3 `==` $\{\}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ S_1 `==` $\{\}$, S_2 `==` $\{\}$, $\Pi \Box \sigma$
S9	S_1 `==` S_2, S_1 `/=` S_2, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ $\blacksquare$

Table 1. Store transformation rules for $solve^{\mathcal{FS}^\mathcal{T}}$, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}} \Pi' \Box \sigma'$

$\mathcal{TOY}$ extension, a general type `elem` is used, but ECL^{i}PS^e sets are only allowed to contain integer elements. It can be easily extended to any other finite domain by means of a bijection into the finite domain of integers.

A constraint solver for the domain $\mathcal{FS}$ is modeled as a function $solve^{\mathcal{FS}}$ which deals with a set Π of $\mathcal{FS}$ constraints. A solver invocation $solve^{\mathcal{FS}}(\Pi)$ returns a finite disjunction of existentially quantified constraint stores composed of constraints and substitutions.

Formally, $solve^{\mathcal{FS}}(\Pi) = \bigvee_{j=1}^{k} \{\exists \overline{Y}_j (\Pi_j \sqsubset \sigma_j) \mid \Pi_j \sqsubset \sigma_j \in SF_{\mathcal{FS}}(\Pi)$, where $\overline{Y}_j = var(\Pi_j \sqsubset \sigma_j) \setminus var(\Pi)\}$. Alternative constraint stores which are returned by solver invocations are usually explored in sequential order using backtracking.

The glass-box solver $\mathcal{FS}^\mathcal{T}$ has been formalized using a store transformation system. Its rules are shown in Table 1, where a given store $\Pi \Box \sigma$ is transformed into another store $\Pi' \Box \sigma'$ in one rewriting step $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}} \Pi' \Box \sigma'$. Rules from **S1** to **S8** infer new constraints in order to improve the performance of the ECL^{i}PS^e solver, and generate equalities and disequalities that may help to anticipate failure by means of rule **S9**. In some cases, such as rules **S4** and **S8**, some constraints can be removed, since both rules infer concrete values to variables. Rule **S1** propagates the information S_1 `/=` $\{\}$,$\ldots$, S_n `/=` $\{\}$ when the constraint `domainSets` $[S_1, \ldots, S_n]$ `lb ub` is processed and `lb` is not empty. The remaining rules are similar. These rules complement the rules already existing in ECL^{i}PS^e, especially for anticipating failure by handing disequalities. More rules based on set theory could be added, as for example `union` S_1 S_2 S_3, $\Pi \Box \sigma \Vdash_{\mathcal{FS}^\mathcal{T}}$ `subset` S_1 S_3, `subset` S_2 S_3.

As semantic results, we present soundness and limited completeness of the glass box $\mathcal{FS}^\mathcal{T}$ solver. Although it is out of the scope of this paper, well-typed solutions $WTSol_\mathcal{P}(G) \subseteq Sol_\mathcal{P}(G)$ are those solutions for which type judgements can be deduced from type assumptions in Σ. Completeness can only be guaranteed for well-typed solutions.

Theorem 1. *The store transformation system with transition relation $\Vdash_{\mathcal{FS}^\mathcal{T}}$ is finitely branching and terminating, and:*

1. *solve*$^{\mathcal{FS}^{\mathcal{T}}}(\Pi)$ *is finite for any finite* $\Pi \subseteq APCon_{\mathcal{FS}^{\mathcal{T}}}$.
2. *(Soundness)* $Sol_{\mathcal{FS}}(\Pi) \supseteq \bigcup\{Sol_{\mathcal{FS}}(\exists\overline{Y}_j(\Pi_j \sqcap \sigma_j)) \mid \Pi_j \sqcap \sigma_j \in SF_{\mathcal{FS}^{\mathcal{T}}}(\Pi)\}$.
3. *(Limited completeness)* $WTSol_{\mathcal{FS}}(\Pi) \subseteq \bigcup\{WTSol_{\mathcal{FS}}(\exists\overline{Y}_j(\Pi_j \sqcap \sigma_j)) \mid \Pi_j \sqcap \sigma_j \in SF_{\mathcal{FS}^{\mathcal{T}}}(\Pi)\}$.

Regarding the black-box solver $\mathcal{FS}^E$, we can assume that *solve*$^{\mathcal{FS}^E}$ reduces primitive constraints to a *solved form*, in the sense that they can not be further reduced. Moreover, we assume that *solve*$^{\mathcal{FS}^E}$ is sound, and the completeness property may fail for some choices of $\Pi \subseteq APCon_{\mathcal{FS}}$, hence completeness in previous theorem is limited by $\mathcal{FS}^E$.

3.3 The Mediatorial Domain $\mathcal{M}$

The $\mathcal{FD}$ domain already existing in $\mathcal{TOY}$ can be joined with the new $\mathcal{FS}$ domain by means of the *amalgamated sum* of both domains, defined as a new domain $\mathcal{S} = \mathcal{FD} \oplus \mathcal{FS}$ with signature $\langle TC,\ SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}},\ DC,\ DF,\ SPF_{\mathcal{FD}} \cup SPF_{\mathcal{FS}} \rangle$. According to [5], $\mathcal{FD}$ and $\mathcal{FS}$ are joinable, and $\mathcal{S}$ is a conservative extension of both domains: for any $p \in SPF^{\mathcal{M}}_{\mathcal{FD}}, \bar{t}_m, t$ patterns, $p^{\mathcal{FD}}\bar{t}_m \to t$ iff $p^{\mathcal{S}}\bar{t}_m \to t$, analogously for $\mathcal{FS}$.

However, this new domain $\mathcal{S}$ has no mechanisms for the communication between both pure domains. In order to establish cooperation between these domains we need a new mediatorial domain, which supplies bridge constraints for communicating $\mathcal{FD}$ and $\mathcal{FS}$. The *mediatorial domain* $\mathcal{M}$ for the communication between $\mathcal{FD}$ and $\mathcal{FS}$ is defined with signature $\langle TC,\ SBT_{\mathcal{M}},\ DC,\ DF,\ SPF_{\mathcal{M}} \rangle$ as follows:

- $SBT_{\mathcal{M}} = \{\texttt{int}, \texttt{set}\} \subseteq SBT_{\mathcal{FD}} \cup SBT_{\mathcal{FS}}$, and $SPF_{\mathcal{M}} = \{$ `#--` $\}$.
- Each set of base values of the mediatorial domain corresponds to a set of base values of each pure domain: $\mathcal{B}^{\mathcal{M}}_{\texttt{set}} = \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$ and $\mathcal{B}^{\mathcal{M}}_{\texttt{int}} = \mathcal{B}^{\mathcal{FD}}_{\texttt{int}}$.
- $SPF_{\mathcal{M}} = \{$`#--` $::$ `int` $\to$ `set` $\to$ `bool`$\}$. The interpretation of the bridge constraint is i `#--`$^{\mathcal{M}} s \to t$, where `#--`$^{\mathcal{M}}$ is a subset of the Cartesian product $\mathbb{Z} \times \mathcal{B}^{\mathcal{FS}}_{\texttt{set}}$, defined to hold iff any of the following cases holds: either s is a set, i is the cardinality of s and $t =$ `true`; or s is a set, i is not the cardinality of s and $t =$ `false`; or $t = \bot$.

We define *solve*$^{\mathcal{M}}$ as a store transformation system, using the same abstract technique for glass-box solvers described previously. Store transformation rules are defined in Table 2. Rule **M1** represents the case of a bridge with a ground set, in which the variable X is bound to the cardinality of the set. In **M2** the cardinality of set variable S is zero and therefore S is bound to the empty set. Rule **M3** considers the case when a set variable has a known number of elements. It can be expressed as a set of `elem` variables with a specific ordering, in order to represent the canonical form of the set. The particular constraints expressing the order of the elements in the set are submitted to the $\mathcal{FS}$ solver. Rules **M4** and **M5** correspond to the case in which the set and the cardinality are ground. If the constraint is satisfied then **M4** is applied else **M5** is applied. Formally, $solve^{\mathcal{M}}(\Pi) = \bigvee\{\exists\overline{Y'}(\Pi' \sqcap \sigma') \mid \Pi' \sqcap \sigma' \in \mathcal{SF}_{\mathcal{M}}(\Pi), \overline{Y'} = var(\Pi' \sqcap \sigma') \setminus var(\Pi)\}$.

M1	X#--u',$\Pi \sqsupset \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \sqsupset \sigma\sigma_1$ if $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$, $X \in Var$ and $\exists u \in \mathbb{Z}^+$ s.t. $u\mathtt{\#--}^{\mathcal{M}}u'$ and $\sigma_1 = \{X \mapsto u\}$.
M2	$u\mathtt{\#--}S,\Pi \sqsupset \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \sqsupset \sigma\sigma_1$ if $u = 0$, $S \in Var$ and $\sigma_1 = \{X \mapsto \{\}\}$
M3	$u\mathtt{\#--}S,\Pi \sqsupset \sigma\ S \vdash_{\mathcal{M}} \Pi\sigma_1 \sqsupset \sigma\sigma_1$ if $u \in \mathbb{Z}^+$, $S \in Var$, $u > 0$ and $\sigma_1 = \{S \mapsto \{X_1, \ldots, X_u\}\}$. $\{X_1 \prec X_2, \ldots, X_{n-1} \prec X_u\}$ is submitted to $\mathcal{FS}$ solver; X_i are fresh variables.
M4	$u\mathtt{\#--}u'\ ,\ \Pi \sqsupset \sigma \vdash_{\mathcal{M}} \Pi \sqsupset \sigma$ if $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$ and $u\mathtt{\#--}^{\mathcal{M}}u'$.
M5	$u\mathtt{\#--}u',\ \Pi \sqsupset \sigma \vdash_{\mathcal{M}} \blacksquare$ if $u \in \mathbb{Z}^+$, $u' \in \mathcal{B}^{\mathcal{FS}}_{\mathtt{set}}$ and $u\mathtt{\#--}^{\mathcal{M}}u'$ does not hold.

Table 2. Store transformation rules for $solve^{\mathcal{M}}$

Example 3. If rule **M3** is applied to $3\mathtt{\#--}X$, where X is a variable, then in this step of computation the constraints $X_1 \prec X_2, X_2 \prec X_3$ are added to $\mathcal{FS}$ store, where X_i are fresh variables, $\sigma_1 = \{X \mapsto \{X_1, X_2, X_3\}\}$, and $3\mathtt{\#--}X,\Pi \sqsupset \sigma \vdash_{\mathcal{M}} \Pi\sigma_1 \sqsupset \sigma\sigma_1$. □

The following theorem ensures that the store transformation system for $\mathcal{M}$-stores can be accepted as a correct specification of a glass-box solver for the domain $\mathcal{M}$.

Theorem 2. *The store transformation system with transition relation $\vdash_{\mathcal{M}}$ is finitely branching and terminating, and:*

1. *$solve^{\mathcal{M}}(\Pi)$ is finite for any finite $\Pi \subseteq APCon_{\mathcal{M}}$.*
2. *(Soundness) $Sol_{\mathcal{M}}(\Pi) \supseteq \bigcup\{Sol_{\mathcal{M}}(\exists\overline{Y}_j(\Pi_j \sqsupset \sigma_j) \mid \Pi_j \sqsupset \sigma_j \in SF_{\mathcal{M}}(\Pi)\}$*
3. *(Completeness) $WTSol_{\mathcal{M}}(\Pi) \subseteq \bigcup\{WTSol_{\mathcal{M}}(\exists\overline{Y}_j(\Pi_j \sqsupset \sigma_j) \mid \Pi_j \sqsupset \sigma_j \in SF_{\mathcal{M}}(\Pi)\}$*

4 Adapting the $CCLNC(\mathcal{C})$ Calculus

The coordination domain that allows the communication among solvers is $\mathcal{C} = \mathcal{M} \oplus \mathcal{H} \oplus \mathcal{FD} \oplus \mathcal{FS}$. Observe that all domains in $\mathcal{C}$ are pairwise joinable according to [5], and communication with $\mathcal{H}$ is automatically performed by variable substitutions. The cooperation mechanism on which $\mathcal{TOY}$ is based is the Cooperative Constrained Lazy Narrowing Calculus, $CCLNC$, that can be instantiated to different constraint domains, and in particular to $\mathcal{C}$, giving $CCLNC(\mathcal{C})$.

A rewriting calculus similar to the one defined in [5] can be adapted to model the behaviour of the $\mathcal{TOY}$ system extended with this coordination domain. In that calculus, goals must handle constraints related to different particular domains. For the coordination domain presented in this paper, goals have the form: $G \equiv \exists\overline{U}.\ P \sqsupset C \sqsupset M \sqsupset H \sqsupset F \sqsupset S$, where:

– $\overline{U}$ is a finite set of *existential* local variables created during the computation.

- P is a set of *productions*. During goal solving, productions are used to obtain values for the variables demanded by the computation, using the goal solving rules for constrained lazy narrowing.
- C is a constraint pool, where constraints are waiting to be solved.
- $M = \Pi_M \,\square\, \sigma_M$ is the *mediatorial store* defined in Section 3.3.
- $H = \Pi_H \,\square\, \sigma_H$ is the *Herbrand store*.
- $F = \Pi_F \,\square\, \sigma_F$ is the *finite domain store*.
- $S = \Pi_S \,\square\, \sigma_S$ is the *finite set store* defined in Section 3.2.

Roughly speaking, the calculus works as follows:

1. Initially, a goal consists of a sequence of constraints placed in C, while the other components are empty.
2. Constraints in C are treated as follows:
   (a) If $\pi \in C$ is an atomic primitive constraint, then π is submitted to the appropriate store, using rules in Table 3.
   (b) Else, π is flattened by $CCLNC(\mathcal{C})$, and is eventually transformed into a conjunction of atomic primitive constraints, possibly using new existential variables. These atomic primitive constraints are placed in C. Some computations are suspended in P by means of lazy narrowing.
3. Eventually, all atomic primitive constraints placed in C are processed. Suspended productions which are not demanded are removed.
4. Finally, a solved goal is obtained: a goal with empty P and C, and where the stores are in solved form.

$CCLNC(\mathcal{C})$ rules that model the behaviour of constrained lazy narrowing ignoring domain cooperation and solver invocation are those presented in [5]. In order to adapt the calculus to the new coordination domain, new rules, defined in Table 3, are needed for handling $\mathcal{M}$ and $\mathcal{FS}$ constraints. Rule **SB** generates mediatorial constraints in M from $\mathcal{FS}$ constraints, while **PP** projects constraints from $\mathcal{FD}$ to $\mathcal{FS}$ and vice versa, and **SC** places the constraints in the corresponding solver. Rule **IE** infers equalities from bridges already existing in M. Rule **IF** infers failure from disequalities detected in bridges in M. Finally, rule **SS** performs the actual black-box set solver invocation. [5] includes rules for $\mathcal{M}$, $\mathcal{H}$ and $\mathcal{FD}$ domains similar to **SS**.

Functions *bridges* and *proj* are used by **SB** and **PP** for obtaining bridges and projections, resp., for each primitive constraint. Given a pool of constraints that includes the atomic primitive constraint π and a mediatorial store with a set of bridge constraints B, we define the function $bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ to generate new bridges for all set variables involved in π as long as they are not already available in B. Projections ($proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$ and $proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$) take place whenever a constraint is posted to its corresponding solver. This process builds mate constraints considering the available bridge constraints, and posts them to the mate solver. Table 4 gives a specification of bridge and projection generation for each constraint π. Each set variable has an associated finite domain variable which represents the cardinality of the set.

SB Set Bridges
$\exists\overline{U}.\ P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \vdash_{\mathbf{SB}} \exists\overline{V}', \overline{U}.\ P \sqcap \pi,\ C \sqcap B', M \sqcap H \sqcap F \sqcap S$
If $\pi \in APCon_{\mathcal{FS}}$ and $\exists\overline{V'}\, B' = bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B) \neq \emptyset$.
PP Propagate Projections
$\exists\overline{U}.\ P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \vdash_{\mathbf{PP}} \exists\overline{V'}, \overline{U}.\ P \sqcap \pi, C \sqcap M \sqcap H \sqcap F' \sqcap S'$
If $\pi \in APCon_{\mathcal{FD}}$ and $\exists\overline{V'}\, \Pi' = proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B) \neq \emptyset$, $F' = F$, and $S' = \Pi', S$,
else $\pi \in APCon_{\mathcal{FS}}$ and $\exists\overline{V'}\, \Pi' = proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B) \neq \emptyset$, $F' = \Pi', F$, and $S' = S$.
SC Submit Constraints
$\exists\overline{U}.\ P \sqcap \pi, C \sqcap M \sqcap H \sqcap F \sqcap S \vdash_{\mathbf{SC}} \exists\overline{U}.\ P \sqcap C \sqcap M' \sqcap H' \sqcap F' \sqcap S'$
Either $\pi \in APCon_{\mathcal{M}}$, then $M' = \pi, M$, $H' = H$, $F' = F$, and $S' = S$,
or $\pi \in APCon_{\mathcal{FD}}$, then $M' = M$, $H' = H$, $F' = \pi, F$, and $S' = S$,
or $\pi \in APCon_{\mathcal{FS}}$, then $M' = M$, $H' = H$, $F' = F$, and $S' = \pi, S$,
or $\pi \in \{\texttt{==},\texttt{/=}\}$ and $\pi \notin APCon_{\mathcal{FD}}$, $\pi \notin APCon_{\mathcal{FS}}$
then $M' = M$, $H' = \pi, H$, $F' = F$, and $S' = S$.
IE Infer Equalities
$\exists\overline{U}.\ P \sqcap C \sqcap (I_1\texttt{\#--}S,\ I_2\texttt{\#--}S,\ \Pi_M \sqcap \sigma_M) \sqcap H \sqcap F \sqcap S \vdash_{\mathbf{IE}}$
$\exists\overline{U}.\ P \sqcap C \sqcap (I_1\texttt{\#--}S,\ \Pi_M \sqcap \sigma_M) \sqcap H \sqcap I_1\ \texttt{==}\ I_2,\ F \sqcap S$
IF Infer Failure
$\exists\overline{U}.\ P \sqcap C \sqcap (I_1\texttt{\#--}S_1,\ I_2\texttt{\#--}S_2,\ \Pi_M \sqcap \sigma_M) \sqcap H \sqcap (I_1\ \texttt{/=}\ I_2,\ \Pi_F \sqcap \sigma_F) \sqcap$
$(S_1\ \texttt{==}\ S_2,\ \Pi_S \sqcap \sigma_S) \vdash_{\mathbf{IF}} \blacksquare$
SS $\mathcal{FS}^E$ black-box Set Solver Invocation
$\exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap (\Pi_S \sqcap \sigma_S) \vdash_{\mathbf{SS}}$
$\exists\overline{Y'}, \overline{U}.\ P\sigma' \sqcap C\sigma' \sqcap M \star \sigma' \sqcap H \star \sigma' \sqcap F \star \sigma' \sqcap (\Pi'\sigma' \sqcap \sigma_S\sigma')$
If $(\Pi_S \sqcap \sigma_S)$ is not solved, and $solve^{\mathcal{FS}^E}(\Pi_S) = \exists\overline{Y'}(\Pi' \sqcap \sigma')$.
$\star$ is the *application* of σ' to $\Pi \sqcap \sigma$: $(\Pi \sqcap \sigma) \star \sigma' = \Pi\sigma' \sqcap (\sigma\sigma' \restriction vdom(\sigma))$ where $\restriction$ stands for
the restriction of the composition $\sigma\sigma'$ to variables in the domain of σ

Table 3. Store transformation rules for $CCLNC(\mathcal{C})$

Example 4. Let us see how this calculus is applied to Example 1. Substitutions are not shown in the goal to avoid overloading the notation. Initially,

$$\emptyset \sqcap \overbrace{\texttt{L==[S1,S2]}}^{\pi_1}, \overbrace{\texttt{domainSets L \{\} \{1,2,3,4,5\}}}^{\pi_2}, \overbrace{\texttt{atMostOne L}}^{\pi_3}, \texttt{labelingSets L} \sqcap \emptyset \sqcap \emptyset \sqcap \emptyset \sqcap \emptyset$$

The constraint π_1 is sent to the $\mathcal{H}$ store and the $\mathcal{H}$ solver is invoked producing the substitution $\sigma_1 = \{\texttt{L} \mapsto \texttt{[S1,S2]}\}$. Next, π_2 is processed using rules of Table 3: First, rule **SB** generates the bridge constraints `I1#--S1` and `I2#--S2` with `I1` and `I2` fresh variables, considering the function $bridges^{\mathcal{FS}\to\mathcal{FD}}$ defined in Table 4. Next, rule **PP** builds $\mathcal{FD}$ constraints corresponding to the projection of π_2:

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi_2, \emptyset) = \{\texttt{I1\#--S1, I2\#--S2}\} = B$$
$$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi_2, B) = \{\texttt{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5}\}$$

π_2 is sent to the $\mathcal{FS}$ store, rule **SC**. The goal is then transformed to:

$$\emptyset \sqcap \texttt{atMostOne [S1,S2]}, \texttt{labelingSets [S1,S2]} \sqcap \texttt{I1\#--S1}, \texttt{I2\#--S2} \sqcap \emptyset \sqcap \texttt{0\#<=I1,I1\#<=5,0\#<=I2,I2\#<=5} \sqcap \pi_2$$

π	$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$	$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi, B)$
S_1 `==` S_2	{`I`$_i$`#--`S_i | S_i has no bridge in B, I_i fresh} (Range for i depends on the variables in π)	{I_1 `==` I_2}
`domainSets L=[`$S_1, \ldots, S_n$`] s s'` and `s, s'`$\in \mathcal{B}^{\mathcal{FS}}_{\mathsf{set}}$		{`c #<=` I_i, I_i `#<= c'` $\mid 1 \leq i \leq n$ and `c#--`$^{\mathcal{M}}$`s, c'#--`$^{\mathcal{M}}$`s'`}
`subset` S_1 S_2		{I_1 `#<=` I_2}
`intersect` S_1 S_2 S_3		{I_3 `#<= min{`I_1`,`I_2`}`, I_1`#>=0`, I_2`#>=0`}
`union` S_1 S_2 S_3		{I_3`#<=`I_1`#+`I_2, `max{`I_1`,`I_2`}#<=`I_3}

π	$bridges^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$	$proj^{\mathcal{FD}\to\mathcal{FS}}(\pi, B)$
I_1 `#<` I_2 (or `#>` or `/=`)	{}	{S_1 `/=` S_2 | I_1`#--`S_1, I_2`#--`$S_2 \in B$}

Table 4. Computing Bridges and Projections from $\mathcal{FS}$ to $\mathcal{FD}$ and from $\mathcal{FD}$ to $\mathcal{FS}$.

Now π_3 is flattened, producing new atomic primitive constraints:

$$\exists\ \texttt{S12,I12}.\emptyset \ \square\ \overbrace{\texttt{3\#--S1}}^{\pi_4},$$
$$\overbrace{\texttt{3\#--S2}}^{\pi_5}, \overbrace{\texttt{intersect S1 S2 S12}}^{\pi_6}, \overbrace{\texttt{I12\#--S12}}^{\pi_7}, \overbrace{\texttt{I12 \#<= 1}}^{\pi_8}, \overbrace{\texttt{labelingSets [S1,S2]}}^{\pi_9} \ \square$$
$$\texttt{I1\#--S1, I2\#--S2} \ \square\ \emptyset \ \square\ \texttt{0\#<=I1, I1\#<=5, 0\#<=I2, I2\#<=5} \ \square\ \pi_2$$

π_4 is sent to the mediatorial store, and $solve^{\mathcal{M}}$ is invoked. Applying rule **M3** from Table 2, a substitution $\sigma_2 = \{\texttt{S1} \mapsto \{\texttt{E11,E12,E13}\}\}$ and $\mathcal{FS}$ constraints `E11`$\prec$`E12` and `E12`$\prec$`E13` are obtained. The same rule applies to π_5 producing the substitution σ_3 $= \{\texttt{S1} \mapsto \{\texttt{E21,E22,E23}\}\}$ and $\mathcal{FS}$ constraints `E21`$\prec$`E22` and `E22`$\prec$`E23`.
Next, the constraint π_6 is processed in a similar way to π_2, generating a new bridge and new $\mathcal{FD}$ constraints:

$$bridges^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B) = \{\texttt{I12\#--S12}\} = B'$$
$$proj^{\mathcal{FS}\to\mathcal{FD}}(\pi_6, B'') = \{\texttt{I12\#<=min(I1,I2), I1\#>=0, I2\#>=0}\}, \text{ where } B'' = B \cup B'$$

Then rule **SC** is applied to π_7, π_8 and π_9, submitting them to the $\mathcal{M}$, $\mathcal{FD}$ and $\mathcal{FS}$ stores, respectively. Finally, P and C are empty.

$\exists$ `S12,I12`.$\emptyset$ $\square$ $\emptyset$ $\square$ `I1#--S1, I2#--S2`, π_4, π_5, π_7 $\square$ $\emptyset$ $\square$ π_8, `0#<=I1, I1#<=5, 0#<=I2, I2#<=5, I12#<=min(I1,I2), I1#>=0, I2#>=0` $\square$ π_2, π_6, `E11`$\prec$`E12`, `E12`$\prec$`E13`, `E21`$\prec$`E22`, `E22`$\prec$`E23`, π_9.

Taking into consideration `I1#--S1`, π_4, `I2#--S2`, and π_5, it is possible to infer the equalities `I1 == 3` and `I2 == 3` applying rule **IE** of Table 3 twice. These equalities are posted to the $\mathcal{H}$ store, producing the substitution $\sigma_4 = \{\texttt{I1} \mapsto \texttt{3}, \texttt{I2} \mapsto \texttt{3}\}$:

$\exists$ `S12,I12`.$\emptyset$ $\square$ $\emptyset$ $\square$ `3#--S1, 3#--S2`, π_7 $\square$ $\emptyset$ $\square$ π_8,`I12#<=3` $\square$ π_2, π_6, `E11`$\prec$`E12`, `E12`$\prec$`E13`,`E21`$\prec$`E22`, `E22`$\prec$`E23`,π_9.

If any store is not in solved form yet, its respective solver is appropriately invoked. In $\mathcal{FS}$, `labelingSets` enumerates all ground values for `S1` and `S2`. A first attempt is `S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {1,2,3}, but this substitution does not satisfy π_6, π_7 and π_8. The rest of the valuations are checked on backtracking until a solution is obtained, as for example `S1` $\mapsto$ {1,2,3}, `S2` $\mapsto$ {1,4,5}. $\square$

The *set of solutions* $Sol_{\mathcal{P}}(G)$ of $G \equiv \exists\overline{U}.\ P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap S$ w.r.t. a program $\mathcal{P}$ includes all those $\mu \in Val_{\mathcal{C}}$ such that there is some $\mu' \in Val_{\mathcal{C}}$ verifying $\mu' =_{\setminus \overline{U}} \mu$ and $\mu' \in Sol_{\mathcal{P}}(P \sqcap C \sqcap M \sqcap H \sqcap F \sqcap S)$, which holds iff the following two conditions are satisfied:

1. $\mu' \in Sol_{\mathcal{P}}(P \sqcap C)$. By definition, this means $\mathcal{P} \vdash_{CRWL(\mathcal{C})} (P \sqcap C)\mu'$, which is equivalent to $\mathcal{P} \vdash_{CRWL(\mathcal{C})} P\mu'$ and $\mathcal{P} \vdash_{CRWL(\mathcal{C})} C\mu'$. This notation refers to the existence of proofs in the instance $CRWL(\mathcal{C})$ of the *Constrained Rewriting Logic* $CRWL$, whose inference rules can be found in [11].
2. $\mu' \in Sol_{\mathcal{C}}(M \sqcap H \sqcap F \sqcap S)$, which is equivalent to $\mu' \in Sol_{\mathcal{C}}(M) \cap Sol_{\mathcal{C}}(H) \cap Sol_{\mathcal{C}}(F) \cap Sol_{\mathcal{C}}(S)$, where $Sol_{\mathcal{D}}$ for a domain $\mathcal{D}$ was defined in Section 3.

In this work, we present the following semantic results for the cooperative goal solving calculus $CCLNC(\mathcal{C})$: *soundness* and *limited completeness* with respect to *Constraint ReWriting Calculus* $CRWL(\mathcal{D})$ [11], which provides a declarative semantics for $CFLP(\mathcal{D})$ programs.

Soundness result ensures that the solved forms obtained as computed answers for an initial goal using the rules of the cooperative goal solving calculus are indeed semantically valid answers of G.

Theorem 3 (Soundness). *Assume a $CFLP(\mathcal{C})$-program $\mathcal{P}$, a goal G for $\mathcal{P}$, and a solved goal S such that $G \vdash^*_{\mathcal{P}} S$. Then, $Sol_{\mathcal{C}}(S) \subseteq Sol_{\mathcal{P}}(G)$.*

Completeness result ensures that, under some limitations, any well-typed solution can be obtained by $CCLNC(\mathcal{C})$.

Theorem 4 (Limited Completeness). *Let G be a goal for a program $\mathcal{P}$ and $\mu \in WTSol_{\mathcal{P}}(G)$ a well-typed solution. Assume that neither $\mathcal{P}$ nor G are* unsafe*. Then, unless there are* unsafe *rule applications, a $CCLNC(\mathcal{C})$ computation $G \vdash^*_{\mathcal{P}} S$ can be found, ending with a goal in solved form S such that $\mu \in WTSol_{\mathcal{C}}(S)$, i.e., $WTSol_{\mathcal{C}}(S) \supseteq WTSol_{\mathcal{P}}(G)$.*

A program $\mathcal{P}$ or goal G is called *unsafe* if they have free occurrences of higher-order variables, or 'opaque decompositions' that can produce ill-typed goals [5]. *Unsafe* rule applications are some applications of **DC** rule from constrained lazy narrowing (in [5]) and rules involving incomplete black-box solvers invocations.

5 Implementation and Experiments

A prototype of this extension has been added to the $\mathcal{TOY}$ system. Not all features described in this paper are included: set constraints are directly passed to the ECL^{i}PS^e solver ($\mathcal{FS}^{\mathcal{T}}$ is not implemented yet). Nevertheless, the preliminary results obtained are promising, and we expect to improve them when the full implementation is available.

The extension of $\mathcal{TOY}$ with ECL^{i}PS^e domains has been designed to keep it as simple as possible. An ECL^{i}PS^e program executes in a different process acting as a server, accepting and executing requests from the $\mathcal{TOY}$ process. During the

execution of a $\mathcal{TOY}$ goal, for every constraint π in $APCon_{\mathcal{FS}}$ that needs to be evaluated, a request is issued to the ECL^{i}PS^e server. The server evaluates π and returns the result to the $\mathcal{TOY}$ process. For communicating both processes, standard input/output library predicates have been used operating on a pipe that interconnects them.

The implementation of this extension of $\mathcal{TOY}$ has been developed for working in two modes: The first mode, referred to as *interactive*, is oriented to model reasoning, i.e., when the model can be modified during solving. It is based on an interactive communication between $\mathcal{TOY}$ and ECL^{i}PS^e. When $\mathcal{TOY}$ requires a constraint to be solved in ECL^{i}PS^e, $\mathcal{TOY}$ posts it and blocks until ECL^{i}PS^e server returns an answer. This approach takes advantage of all features of the $\mathcal{TOY}$ language with the new $\mathcal{FS}$ domain.

The second, *batch* mode, is intended for classic CP applications, where constraints are first specified and then posted and solved. This approach delays the computation of some or all constraints of the current goal, sending them to ECL^{i}PS^e at the end of the $\mathcal{TOY}$ narrowing. This mode avoids the interactive communication between both processes that may slow down the execution of goals. However, communication is inevitable when more answers are demanded for a given goal. The backtracking mechanism forces interactive communication between both processes, even though the constraints are sent in batch mode.

Several issues have been addressed in order to make the system work. The most relevant issue is related to the cooperation between constraint domains. Since ECL^{i}PS^e includes a finite domain tightly integrated with the sets domain, the finite domain already existing in $\mathcal{TOY}$ (provided by the SICStus implementation) has been replaced by the ECL^{i}PS^e finite domain.

The preliminary implementation has been tested with two well-known examples in the literature about integer sets solvers, taken from the ECL^{i}PS^e web page [3]: *Steiner triplets* and *Social Golfers* problems. $\mathcal{TOY}$ formulations are accessible in `http://gpd.sip.ucm.es/sonia/systems.html`. The *Steiner Triplets Problem* of order `n` consists of finding a set of `n(n-1)/6` triples of distinct integer elements in `{1,2,...,n}` such that any two triples have at most one element in common. The second example is *The Social Golfers Problem.* It consists of trying to schedule `g*p` golfers into `g` groups of `p` players over `w` weeks, such that no golfer plays in the same group with any other golfer more than just once.

These programs have been executed with different parameters, as shown in column **Bench.** of Table 5, which shows the results for the computation of all solutions or until failure when there are no solutions for the input arguments. Symmetries are not removed, in order to have the same behavior as the original ECL^{i}PS^e programs.

We have executed these examples in $\mathcal{TOY}$ and compared their performance with the existing implementation in ECL^{i}PS^e. Both examples have been executed in a system with an Intel© Core$^{\text{TM}}$ i7-740QM processor (4 cores) at 1.73 GHz with 3 GB of physical memory and running Ubuntu 10.10. Experiments marked with a hyphen in their execution time have reached a time out of 3,600 seconds. **ECL^{i}PS^e** column contains the time in milliseconds spent by

Bench.	ECL^{i}PS^e	**inter**	**SU**	**batch**	**SU**	$\mathcal{TOY}$ **ecl**	**SU**
steiner3	0	0	n/a	0	n/a	0	n/a
steiner4	0	10	n/a	10	n/a	10	n/a
steiner5	10	50	0.20	30	0.33	10	1.00
steiner6	670	1 280	0.52	810	0.83	680	0.99
steiner7	266 210	1 869 850	0.14	1 434 930	0.19	262 480	1.01
golf4_2_2	40	360	0.11	90	0.44	50	0.80
golf5_2_2	50	410	0.12	100	0.50	50	1.00
golf6_2_2	60	540	0.11	130	0.46	60	1.00
golf7_2_2	60	700	0.08	140	0.42	70	0.85
golf8_2_2	70	790	0.08	190	0.36	70	1.00
golf9_2_2	70	1 010	0.06	220	0.31	80	0.87
golf10_2_2	80	1 400	0.05	270	0.29	80	1.00
golf3_3_2	51 138	724 550	0.07	928 880	0.05	50 630	1.01
golf4_3_2	1 043 113	-	n/a	-	n/a	1 046 230	0.99

Table 5. Results of experiments for all solutions (execution times in milliseconds).

the original programs[1] (taken from ECL^{i}PS^e site [3]). Column **inter** is the time in milliseconds taken by $\mathcal{TOY}$ interactive mode. Column **batch** contains the time spent by $\mathcal{TOY}$ batch mode. Columns named **SU** contain the speedup of each mode w.r.t. the ECL^{i}PS^e column. n/a stands for speedups which are not computable because any of the values is zero or not available.

In addition to the $\mathcal{TOY}$ modes, it has been measured the time that ECL^{i}PS^e takes in solving the set of constraints sent by $\mathcal{TOY}$, but removing any overhead produced by narrowing and inter-process communication. This has been done by generating an ECL^{i}PS^e Prolog program that contains all atomic primitive constraints created by $\mathcal{TOY}$ in the narrowing process. This is shown in Table 5 under $\mathcal{TOY}$ **ecl** heading.

As it can be observed in Table 5, $\mathcal{TOY}$ batch mode is faster than the interactive mode, in which lazy narrowing computation in $\mathcal{TOY}$ is interleaved with constraint solving in ECL^{i}PS^e. The former case does not require to establish a communication from the ECL^{i}PS^e server to $\mathcal{TOY}$ every time a constraint is posted, and therefore computation in ECL^{i}PS^e is performed without interruption for communicating intermediate results to $\mathcal{TOY}$. Batch mode in these particular examples is very appropriate, since the structure of the solution is first generated in $\mathcal{TOY}$ together with all constraints, which are solved in a second step, in a similar way usual CP problems are specified and solved.

It is remarkable that in all cases $\mathcal{TOY}$ **ecl** takes approximately the same time than the original ECL^{i}PS^e program, in particular for `golf4_3_2`, that is a time consuming experiment and generates a very high number of solutions. These results show that $\mathcal{TOY}$ does not introduce a noticeable overhead w.r.t. the tests directly performed in ECL^{i}PS^e.

[1] Social golfers ECL^{i}PS^e program has been modified in order to make it comparable to the $\mathcal{TOY}$ program. The ECL^{i}PS^e website version program does not perform set labeling, producing very poor results.

6 Conclusions and Future Work

This paper presented a theoretical model for cooperation of the pure domains: Herbrand, $\mathcal{FD}$ and $\mathcal{FS}$ in the $\mathcal{TOY}$ system, by means of a mediatorial domain $\mathcal{M}$ for allowing communication among domains. Domain $\mathcal{FS}$ has been designed starting from an existing set domain in the ECL^{i}PS^e system, extended with several transformation rules to infer additional information. We have proved that this model is sound and complete, although completeness of $\mathcal{FS}$ is guaranteed as far as permitted by the completeness of the underlying black-box solver. The prototype system $\mathcal{TOY}$ with ECL^{i}PS^e external solvers has been experimentally tested with promising results.

As future work, we plan to complete the implementation of the system with the $\mathcal{FS}^{\mathcal{T}}$ solver presented in this paper, and perform a thorough experimental evaluation of the system. We are interested as well in the coordination of these domains with the domain $\mathcal{R}$ over real numbers.

References

1. F. Azevedo. Cardinal: A finite sets constraint solver. *Constraints*, 12:93–129, 2007.
2. F. Bergenti, A. Dal Palú, and G. Rossi. Integrating finite domain and set constraints into a set-based constraint language. *Fundam. Inf.*, 96:227–252, 2009.
3. *ECL^{i}PS^e* Web Site. `http://eclipseclp.org/`.
4. S. Estévez, A. Fernández, and F. Sáenz. Cooperation of the Finite Domain and Set Solvers in $\mathcal{TOY}$. In P. Lucio, G. Moreno, and R. Peña, editors, *In Proc. Prole'09*, pages 217–226, Spain, 2009.
5. S. Estévez-Martín, A. J. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez, and R. del Vado Vírseda. On the Cooperation of the Constraint Domains $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ in *CFLP*. *TPLP*, 9:415–527, 2009.
6. A. J. Fernández and P. M. Hill. An interval constraint branching scheme for lattice domains. *J. UCS*, 12(11):1466–1499, 2006.
7. M. Gavanelli, E. Lamma, P. Mello, and M. Milano. Dealing with incomplete knowledge on clp(fd) variable domains. *ACM Trans. Program. Lang. Syst.*, 27:236–263, March 2005.
8. C. Gervet and P. Van Hentenryck. Length-lex ordering for set csps. In *AAAI*, 2006.
9. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. PLILP'93*, volume 714 of *LNCS*, pages 184–200. Springer, 1993.
10. F. López-Fraguas, M. Rodríguez-Artalejo, and R. del Vado-Virseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *PPDP'04*, pages 43–54. ACM Press, 2004.
11. F. López-Fraguas, M. Rodríguez-Artalejo, and R. del Vado Vírseda. A new generic scheme for functional logic programming with constraints. *Higher-Order and Symbolic Computation*, 20(1/2):73–122, 2007.
12. A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *CP'04. Vol. 3258*, LNCS, pages 604–618. Springer, 2004.
13. A. Sadler and C. Gervet. Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics*, 14(1):23–67, 2008.
14. Toy Web Site. `http://toy.sourceforge.net/`.