Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 188 (2007) 37–51

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming

S. Estévez-Martín[a,1] A. J. Fernández[b,2]
T. Hortalá-González[a,1] M. Rodríguez-Artalejo[a,1]
F. Sáenz-Pérez[a,1] R. del Vado-Vírseda[a,1]

[a] *Departamento de Sistemas Informáticos y Programación*
Universidad Complutense de Madrid
Madrid, Spain

[b] *Departamento de Lenguajes y Ciencias de la Computación*
Universidad de Málaga
Málaga, Spain

Abstract

This paper presents a proposal for the cooperation of solvers in constraint functional logic programming, a quite expressive programming paradigm which combines functional, logic and constraint programming using constraint lazy narrowing as goal solving mechanism. Cooperation of solvers for different constraint domains can improve the efficiency of implementations since solvers can take advantage of other solvers' deductions. We restrict our attention to the cooperation of three solvers, dealing with syntactic equality and disequality constraints, real arithmetic constraints, and finite domain ($\mathcal{FD}$) constraints, respectively. As cooperation mechanism, we consider to propagate to the real solver the constraints which have been submitted to the $\mathcal{FD}$ solver (and viceversa), imposing special communication constraints to ensure that both solvers will allow the same integer values for all the variables involved in the cooperation.

Keywords: Cooperating Solvers, Constraints, Functional Logic Programming, Lazy Narrowing, Implementation.

1 Introduction

Cooperation of different solvers for Constraint Programming (shortly CP) has been widely investigated during the last years [4], aiming at the solution of hybrid problems that cannot be handled by a single solver and also at improvements of efficiency, among other things. On the other hand, the Functional and Logic Programming styles (FP and LP, resp.) support a clean declarative semantics as well as powerful

[1] Author partially supported by projects TIN2005-09207-C03-03 and S-0505/TIC0407

[2] Author partially supported by projects TIN2004-7943-C04-01 and TIN2005-08818-C04-01

doi:10.1016/j.entcs.2007.05.037

program construction facilities. The CLP scheme for Constraint Logic Programming, started by a seminal paper by Jaffar and Lassez, provides a combination of CP and LP which has proved very practical for CP applications [7]. Adding a FP dimension to CLP has led to various proposals of $CFLP$ schemes for Constraint Functional Logic Programming, developed since 1991 and aiming at a very expressive combination of CP, higher-order lazy FP and LP.

Both CLP and $CFLP$ are schemes that can be instantiated by means of different constraint domains and solvers. This paper presents a proposal for solver cooperation in $CFLP$, more precisely in an instance of the $CFLP$ scheme as presented in [9,2,8] which is implemented in the $\mathcal{TOY}$ language and system [1]. The solvers whose cooperation is supported are: a solver for the Herbrand domain $\mathcal{H}$ supporting syntactic equality and disequality constraints; a solver for the domain $\mathcal{FD}$, which supports finite domain constraints over the set of integer numbers $\mathbb{Z}$; and a solver for the domain $\mathcal{R}$, which supports arithmetic constraints over the set of real numbers $\mathbb{R}$. This particular combination has been chosen because of the usefulness of $\mathcal{H}$ constraints for dealing with structured data and the important role of hybrid $\mathcal{FD}$ and $\mathcal{R}$ constraints in many practical CP applications [4].

$\mathcal{TOY}$ has been implemented on top of SICStus Prolog [15], using the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog code for the $\mathcal{H}$ solver. $CFLP$ goal solving takes care of evaluating calls to program defined functions by means of lazy narrowing, and decomposing hybrid constraints by introducing new local variables. Eventually, pure $\mathcal{FD}$ and $\mathcal{R}$ constraints arise, which must be submitted to the respective solvers. Our proposal for solver cooperation is based on the communication between the $\mathcal{FD}$ and $\mathcal{R}$ solvers by means of special communication constraints called *bridges*. A bridge `u #== v` constrains `u::int` and `v::real` to take the same integer value. Our system keeps bridges in a special store and uses them for two purposes, namely *binding* and *propagation*. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end of the bridge becomes a numeric value. Propagation is a more complex operation which takes place whenever a pure constraint is submitted to the $\mathcal{FD}$ or $\mathcal{R}$ solver. At that moment, propagation rules relying on the available bridges are used for building a mate constraint which is submitted to the mate solver (think of $\mathcal{R}$ as the mate of $\mathcal{FD}$ and viceversa). Propagation enables each of the two solvers to take advantage of the computations performed by the other. In order to maximize the opportunities for propagation, the $CFLP$ goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules. Obviously, independent computing of solvers remains possible.

The rest of the paper is organized as follows. Section 2 recalls the essentials of $CFLP$ programming and presents a $CFLP$ program which solves a generic problem illustrating $\mathcal{H}+\mathcal{FD}+\mathcal{R}$ cooperation. Section 3 presents a formal description of cooperative goal solving by means of constraint lazy narrowing enhanced with rules for creation of bridges and propagation of mate constraints. Section 4 presents some details of our current implementation of cooperative goal solving in the $\mathcal{TOY}$ system, as well as performance results based on the program from Section 2, showing that propagation of mate constraints via bridges leads to significant speedups of execution time. Section 5 includes a summary of conclusions, a brief discussion of related work and some hints to planned future work.

2 $CFLP$ Programming

In this section, we recall the essentials of the $CFLP$ scheme [9,2,8] for lazy Constraint Functional Logic Programming, which serves as a logical and semantic framework for our proposal of cooperation of solvers.

2.1 The Constraint Domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$

We assume a *universal signature* $\Sigma = \langle DC,\ FS \rangle$, where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are countably infinite and mutually disjoint sets of *constructor symbols* and *function symbols*, indexed by arities. Functions are further classified into domain dependent *primitive functions* $PF^n \subseteq FS^n$ and user *defined functions* $DF^n = FS^n \setminus PF^n$ for each $n \in \mathbb{N}$. We consider a special symbol $\perp$, intended to denote an *undefined value* and we assume the Boolean constants $true, false \in DC^0$. We also consider a countably infinite set Var of *variables* and a set $\mathcal{U}$ of *primitive elements* (as e.g. the set $\mathbb{Z}$ of integer numbers or the set $\mathbb{R}$ of real numbers).

An *expression* $e \in Exp$ has the syntax $e ::= u \mid X \mid h \mid (e\, e_1 \ldots e_m)$, where $u \in \mathcal{U}$, $X \in Var$ and $h \in DC \cup FS$, ($\overline{e_m}$ abbreviates $e_1 \ldots e_m$). The following classification of expressions is useful: $(X\, \overline{e_m})$, with $X \in Var$ and $m \geq 0$, is called a *flexible expression*, while $u \in \mathcal{U}$ and $(h\ \overline{e_m})$ with $h \in DC \cup FS$ are called *rigid expressions*. Moreover, a rigid expression $(h\, \overline{e_m})$ is called *active* iff $h \in FS^n$ and $m \geq n$, and *passive* otherwise. Another important subclass of expressions is the set of *patterns* $t \in Pat$, whose syntax is defined as $t ::= u \mid X \mid (c\, \overline{t_m}) \mid (f\ \overline{t_m})$, where $u \in \mathcal{U}$, $X \in Var$, $c \in DC^n$ with $m \leq n$, and $f \in FS^n$ with $m < n$. We also consider *substitutions* as mappings σ, θ from variables to patterns, and by convention, we write $e\,\sigma$ instead of $\sigma(e)$ for any $e \in Exp$, and $\sigma\theta$ for the composition of σ and θ.

A *constraint domain* provides a set of specific primitive elements $\mathcal{U}$, along with certain primitive functions $p \in PF^n$ operating upon them. *Atomic constraints* over a given constraint domain $\mathcal{D}$ can have the form $\Diamond$ (denoting a constraint trivially true), $\blacklozenge$ (denoting a constraint trivially false) or $p\ \overline{e_n} \rightarrow!\ t$ with $\overline{e_n} \in Exp$ and $t \in Pat$. *Atomic primitive constraints* have the form $\Diamond$, $\blacklozenge$ or $p\ \overline{t_n} \rightarrow!\ t$ with $\overline{t_n}, t \in Pat$. This paper deals with three constraint domains:

- $\mathcal{H}$, the so-called *Herbrand domain*, which supports syntactic equality and disequality constraints over an empty set of primitive elements.
- $\mathcal{FD}$, which supports *finite domain* constraints over $\mathbb{Z}$.
- $\mathcal{R}$, which supports *arithmetic* constraints over $\mathbb{R}$.

Table 1 summarizes the primitive functions available for these domains, and the way they are used for building atomic primitive constraints in practice. We also assume constraint solvers Solver$^{\mathcal{H}}$, Solver$^{\mathcal{FD}}$ and Solver$^{\mathcal{R}}$ associated to these domains. In addition to the constraints just described, we also use a special kind of *communication constraints* built from a new primitive function $equiv :: int \rightarrow real \rightarrow bool$ such that $(equiv\ n\ x)$ returns *true* if x has an integer value equivalent to n, and $false$ otherwise. Constraints of the form $equiv\ e_1\ e_2 \rightarrow!\ true$ will be called *bridges* and abbreviated as $e_1\ \#==\ e_2$ in the sequel. We introduce a constraint domain $\mathcal{M}$ which operates with bridges. The cooperation of a $\mathcal{FD}$ solver and a $\mathcal{R}$ solver via communication bridges can lead to great reductions of the $\mathcal{FD}$ search space, manifesting as significant speedups of the execution time, as we will see in Section 4.

$\mathcal{D}$	Primitive Functions	Abbreviates
$\mathcal{H}$	$seq :: A \rightarrow A \rightarrow bool$	$e_1 == e_2 =_{def} seq\ e_1\ e_2 \rightarrow!\ true$ $e_1 /= e_2 =_{def} seq\ e_1\ e_2 \rightarrow!\ false$
$\mathcal{FD}$	$iseq :: int \rightarrow int \rightarrow bool$	$e_1\ \#= e_2 =_{def} iseq\ e_1\ e_2 \rightarrow!\ true$ $e_1\ \#\backslash= e_2 =_{def} iseq\ e_1\ e_2 \rightarrow!\ false$
	$ileq :: int \rightarrow int \rightarrow bool$	$e_1\ \#< e_2 =_{def} e_2\ ileq\ e_1 \rightarrow!\ false$ $e_1\ \#<= e_2 =_{def} e_1\ ileq\ e_2 \rightarrow!\ true$ $e_1\ \#> e_2 =_{def} e_1\ ileq\ e_2 \rightarrow!\ false$ $e_1\ \#>= e_2 =_{def} e_2\ ileq\ e_1 \rightarrow!\ true$
	$\#+, \#-, \#*, \#/ :: int \rightarrow int \rightarrow int$, $domain :: [int] \rightarrow int \rightarrow int \rightarrow bool$, $belongs :: int \rightarrow [int] \rightarrow bool$, $labeling :: [labelType] \rightarrow [int] \rightarrow bool$	
$\mathcal{R}$	$rleq :: real \rightarrow real \rightarrow bool$	$e_1 < e_2 =_{def} e_2\ rleq\ e_1 \rightarrow!\ false$ $e_1 <= e_2 =_{def} e_1\ rleq\ e_2 \rightarrow!\ true$ $e_1 > e_2 =_{def} e_1\ rleq\ e_2 \rightarrow!\ false$ $e_1 >= e_2 =_{def} e_2\ rleq\ e_1 \rightarrow!\ true$
	$+, -, *, / :: real \rightarrow real \rightarrow real$	
$\mathcal{M}$	$equiv :: int \rightarrow real \rightarrow bool$	$e_1\# == e_2 =_{def} equiv\ e_1\ e_2 \rightarrow!\ true$

Table 1
The Constraint Domains $\mathcal{H}$, $\mathcal{FD}$, $\mathcal{R}$ and $\mathcal{M}$

2.2 Structure of Program Rules

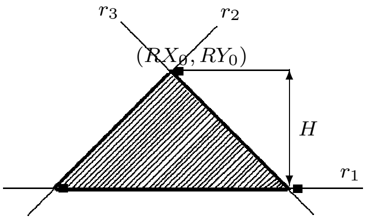
Programs are sets of constrained program rules of the form $f\,\overline{t_n} = r \Leftarrow C$, where $f \in DF^n$, $\overline{t_n}$ is a linear sequence of patterns, r is an expression and C is a finite conjunction $\delta_1, \ldots, \delta_m$ of atomic constraints δ_i for each $1 \leq i \leq m$, possibly including occurrences of defined function symbols. *Predicates* can be modelled as defined functions returning Boolean values, and clauses $p\,\overline{t_n} :- C$ abbreviate rules $p\,\overline{t_n} = true \Leftarrow C$. In practice, $\mathcal{TOY}$ and similar constraint functional logic languages requires program rules to be well-typed in a polymorphic type system.

As a running example for the rest of the paper, we consider a generic program written in $\mathcal{TOY}$ which solves the problem of searching for a $2D$ point lying in the intersection of a discrete grid and a continuous region. Both grids and regions are represented as Boolean functions. They can be passed as parameters because our programming framework supports higher-order programming features.

```
% Discrete versus continuous points:
type dPoint = (int, int)       type cPoint = (real, real)

% Sets and membership:
type setOf A = A -> bool
isIn ::  setOf A -> A -> bool
isIn Set Element = Set Element

% Grids and regions as sets of points:
type grid = setOf dPoints      type region = setOf cPoints

% Predicate for computing intersections of regions and grids:
bothIn::  region -> grid -> dPoint -> bool
bothIn Region Grid (X, Y) :- X #== RX, Y #== RY,
     isIn Region (RX, RY), isIn Grid (X,Y), labeling [ ] [X,Y]
```

We will try the `bothIn` predicate for various square grids and triangular regions of parametrically given sizes, defined as follows:

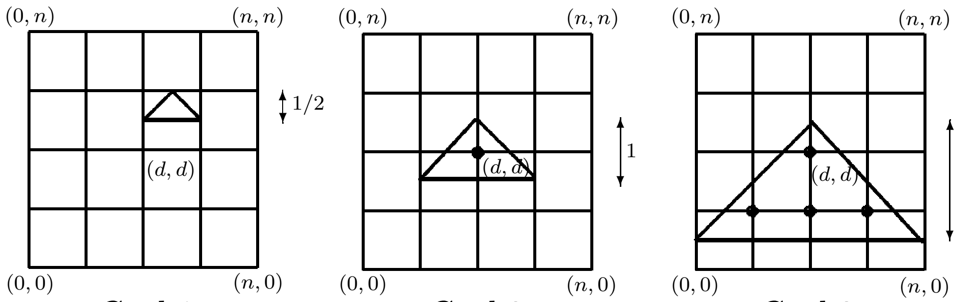
```
% Square grid:
square::  int -> grid
square N (X, Y) :- domain [X, Y] 0 N

% Triangular region:
triangle ::  cPoint -> real -> region
triangle (RX0, RY0) H (RX, RY) :-
    RY >= RY0 - H, RY - RX <= RY0 - RX0, RY + RX <= RY0 + RX0
```



We build an isosceles triangles from a given upper vertex (RX_0, RY_0) and a given height H. The three vertices are (RX_0, RY_0), (RX_0-H, RY_0-H), (RX_0+H, RY_0-H), and the region inside the triangle is enclosed by the lines $r_1 : RY = RY_0 - H$, $r_2 : RY - RX = RY_0 - RX_0$ and $r_3 : RY + RX = RY_0 + RX_0$ and characterized by the conjuntion of the three linear inequalities: $C_1 : RY \geq RY_0 - H$, $C_2 : RY - RX \leq RY_0 - RX_0$ and $C_3 : RY + RX \leq RY_0 + RX_0$. This explains the real arithmetic constraints in the `triangle` predicate.

As an example of goal solving for this program, we fix two integer values d and n such that (d, d) is the middle point of the grid `(square n)`, where $(n + 1)^2$ is the total number of discrete points within the square grid. For instance, we could choose $n = 4$ and $d = 2$. We consider three *goals* computing the intersection of this fixed square grid with three different triangular regions:

- **Goal 1**: `bothIn (triangle (d + 1/2, d+1) 1/2) (square n) (X,Y)`.
  This goal fails.
- **Goal 2**: `bothIn (triangle (d, d+1/2) 1) (square n) (X,Y)`.
  This goal computes one solution for `(X,Y)`, corresponding to the point (d, d).
- **Goal 3**: `bothIn (triangle (d, d+1/2) 2) (square n) (X,Y)`.
  This goal computes four solutions for `(X,Y)`, corresponding to the points (d, d), $(d - 1, d - 1)$, $(d, d - 1)$ and $(d + 1, d - 1)$.



Goal 1 **Goal 2** **Goal 3**

3 Cooperative Goal Solving

Extending the operational semantics given in [9,2] for lazy constraint functional logic programming, we design in this section a goal solving calculus based on constraint lazy narrowing and solver cooperation mechanisms.

3.1 Structure of the Goals

We consider *goals* of the general form $G \equiv \exists\overline{U}.\ P \,\square\, C \,\square\, M \,\square\, H \,\square\, F \,\square\, R$ in order to represent a generic state of the computation with cooperation of solvers over $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$. The symbol $\square$ is interpreted as conjunction.

- $\overline{U}$ is a finite set of local variables in the computation.
- P is a conjunction of so-called *productions* of the form $e_1 \to t_1, \ldots, e_n \to t_n$, where $e_i \in Exp$ and $t_i \in Pat$ for all $1 \le i \le n$. The set of *produced variables* of G is defined as the set $pvar(P)$ of variables occurring in $t_1 \ldots t_n$.
- C is a finite conjunction of constraints to be solved possibly including occurrences of defined functions symbols.
- M is the so-called *communication store* between $\mathcal{FD}$ and $\mathcal{R}$, with primitive bridge constraints involving only variables and integer or real values.
- H is the so-called *Herbrand store*, with strict equality/disequality primitive constraints and an answer substitution with variable bindings.

- F is the so-called *finite domain store*, with finite domain primitive constraints and an answer substitution with integer variable bindings.
- R is the so-called *real arithmetic store*, with primitive real arithmetic constraints and an answer substitution with real variable bindings.

We work with *admissible* goals G satisfying the *goal invariants* given in [9] and such that no variable has more than one bridge in M. We also write $\blacksquare$ to denote an *inconsistent goal.* Moreover, we say that a variable X is a *demanded variable* in a goal G iff X occurs in any of the constraint stores of G (i.e., M, H, F or R), and $\mu(X) \neq \perp$ holds for every solution μ of the corresponding constraint store. For example, `X` is a demanded variable for the finite domain constraint `X#>=3` but not a demanded variable for the strict disequality constraint `s(X)/=0`, where `s` and `0` are constructor symbols. In the sequel, we use the following notations in order to indicate the transformation of a goal by applying a substitution σ and also adding σ to the corresponding store:

- $(P \;\square\; C \;\square\; M \;\square\; H \;\square\; F \;\square\; R)\ @_H\sigma =_{def} (P\sigma \;\square\; C\sigma \;\square\; M\sigma \;\square\; H \upharpoonright \sigma \;\square\; F\sigma \;\square\; R\sigma)$
- $(P \;\square\; C \;\square\; M \;\square\; H \;\square\; F \;\square\; R)\ @_F\sigma =_{def} (P\sigma \;\square\; C\sigma \;\square\; M\sigma \;\square\; H\sigma \;\square\; F \upharpoonright \sigma \;\square\; R\sigma)$
- $(P \;\square\; C \;\square\; M \;\square\; H \;\square\; F \;\square\; R)\ @_R\sigma =_{def} (P\sigma \;\square\; C\sigma \;\square\; M\sigma \;\square\; H\sigma \;\square\; F\sigma \;\square\; R \upharpoonright \sigma)$

where $(\Pi \;\square\; \theta) \upharpoonright \sigma =_{def} \Pi\sigma \;\square\; \theta\sigma$ and $(\Pi \;\square\; \theta)$ stands for H, F or R.

3.2 Cooperative Goal Solving by means of Constrained Lazy Narrowing

The *Constrained Lazy Narrowing Calculus* $CLNC(\mathcal{D})$ is presented in [9] as a suitable computation mechanism for solving goals for $CFLP(\mathcal{D})$ over a single constraint domain $\mathcal{D}$ (e.g. $\mathcal{H}$, $\mathcal{FD}$ or $\mathcal{R}$) and a single constraint solver over the domain $\mathcal{D}$. Now, in order to provide a formal foundation to our proposal for the cooperation of solvers over the constraint domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$, preserving the good properties obtained in the $CFLP(\mathcal{D})$ framework, we have to reformulate the goal transformation rules of the calculus $CLNC(\mathcal{D})$ to deal with the class of goals defined above. We have to distinguish two kinds of rules: rules for constrained lazy narrowing with *sharing* by means of productions (these rules are easily adapted from [9]; see Table 2), and new rules for cooperative constraint solving over the constraint stores.

3.3 Rules for Cooperative Goal Solving

The following three rules describe the process of lazy flattening of non-primitive arguments from constraints in C by means of new productions, the creation of new bridge constraints stored in M with the aim of enabling propagations, and the actual propagation of mate constraints (recall introduction) via bridges, taking place simultaneously with the submission of primitive constraints to the $\mathcal{FD}$ and $\mathcal{R}$ stores.

FC Flatten Constraint

$$\exists\overline{U}.\ P \;\square\; p\ \overline{e_n} \ \to!\ t,\ C \;\square\; M \;\square\; H \;\square\; F \;\square\; R \Vdash_{\mathbf{FC}}$$
$$\exists\overline{V_m},\ \overline{U}.\ \overline{a_m \to V_m},\ P \;\square p\ \overline{t_n} \ \to!\ t,\ C \;\square\; M \;\square\; H \;\square\; F \;\square\; R$$

If some $e_i \notin Pat$, $\overline{a_m}$ are those e_i which are not patterns, $\overline{V_m}$ are new variables,

DC Decomposition

$\exists\overline{U}.\ h\ \overline{e_m} \to h\ \overline{t_m},\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{DC}} \exists\overline{U}.\ \overline{e_m \to t_m},\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R$

CF Conflict Failure $\exists\overline{U}.\ e \to t, P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{CF}} \blacksquare$

if e is rigid and passive, $t \notin Var$, e and t have conflicting roots.

SP Simple Production

$\exists\overline{U}.\ s \to t,\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{SP}} \exists\overline{U'}.\ (P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R)@_H\sigma$

if $s \equiv X \in Var$, $t \notin Var$, $\sigma = \{X \mapsto t\}$ or $s \in Pat$, $t \equiv X \in Var$, $\sigma = \{X \mapsto s\}$; $\overline{U'} \equiv \overline{U} \setminus \{X\}$.

IM Imitation

$\exists X, \overline{U}.\ h\ \overline{e_m} \to X,\ P \,\Box\, C \,\Box\, M \Box H \Box F \Box R \Vdash_{\mathbf{IM}} \exists\overline{X_m}, \overline{U}.\ (\overline{e_m \to X_m},\ P \,\Box\, C \,\Box\, M \Box H \Box F \Box R)\sigma$

if $h\ \overline{e_m} \notin Pat$ is passive, X is a demanded variable and $\sigma = \{X \mapsto h\ \overline{X_m}\}$.

EL Elimination $\exists X, \overline{U}.\ e \to X,\ P \,\Box\, C \,\Box\, M \Box H \Box F \Box R \Vdash_{\mathbf{EL}} \exists\overline{U}.\ P \,\Box\, C \,\Box\, M \Box H \Box F \Box R$

if X does not occur in the rest of the goal.

DF Defined Function

$\exists\overline{U}.\ f\ \overline{e_n a_k} \to t,\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{DF}}$

$\exists X, \overline{Y}, \overline{U}.\ \overline{e_n \to t_n},\ r \to X,\ X\ \overline{a_k} \to t,\ P \,\Box\, C',\ C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R$

if $f \in DF^n$ $(k > 0)$, $t \notin Var$ or t is a demanded variable and $R : f\ \overline{t_n} \to r \Leftarrow C'$ is a fresh variant of a rule in $\mathcal{P}$, with $\overline{Y} = var(R)$ and X are new variables.

PC Place Constraint

$\exists\overline{U}.\ p\ \overline{e_n} \to t,\ P \,\Box\, C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{PC}} \exists\overline{U}.\ \ P \,\Box\, p\ \overline{e_n} \to!\ t,\ C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R$

if $p \in PF^n$ $(k > 0)$, $t \notin Var$ or t is a demanded variable.

Table 2
Rules for Constrained Lazy Narrowing

$p\ \overline{t_n}$ is obtained from $p\ \overline{e_n}$ by replacing each e_i which is not a pattern by V_i.

SB Set Bridges

$$\exists\overline{U}.\ P \,\Box\, p\ \overline{t_n}\ \to!\ t,\ C \,\Box\, M \,\Box\, H \,\Box\, F \,\Box\, R \Vdash_{\mathbf{SB}}$$
$$\exists\overline{V},\ \overline{U}.\ P \,\Box\, p\ \overline{t_n}\ \to!\ t,\ C \,\Box\, M',\ M \,\Box\, H \,\Box\, F \,\Box\, R$$

If $\pi = p\ \overline{t_n}\ \to!\ t$ is a primitive constraint, and
(i) π is a $\mathcal{FD}$ constraint, and $M' = bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, M) \neq \emptyset$ or else
(ii) π is a $\mathcal{R}$ constraint, and $M' = bridges^{\mathcal{R}\to\mathcal{FD}}(\pi, M) \neq \emptyset$.
In both cases, $\overline{V}{=}var(M')\backslash var(M)$ are new variables occurring in the new bridge constraints created by the *bridges* operations described in Tables 3, 4.

SC Submit Constraints

$$\exists\overline{U}.\ P \,\Box\, p\ \overline{t_n}\ \to!\ t,\ C \,\Box M \Box H \Box F \Box R \Vdash_{\mathbf{SC}} \exists\overline{U}.\ P \Box C \Box\ M' \,\Box\, H' \,\Box\, F' \,\Box\, R'$$

If **SB** cannot be used to set new bridges, and one of the following cases applies:

(i) If $p\ \overline{t_n}\ \to!\ t$ is a bridge $u\ \#{==}\ u'$ then $M' = (u\ \#{==}\ u',\ M)$, $H' = H$, $F' = F$ and $R' = R$.

(ii) If $p\ \overline{t_n}\ \to!\ t$ is a primitive Herbrand constraint $seq\ t_1\ t_2\ \to!\ t$ then $M' = M$, $H' = (seq\ t_1\ t_2\ \to!\ t,\ H)$, $F' = F$ and $R' = R$.

(iii) If $p\ \overline{t_n}\ \to!\ t$ is a primitive $\mathcal{FD}$ constraint π then $M' = M$, $H' = H$, $F' = (\pi,\ F)$ and $R' = (R'',\ R)$, where $R'' = propagations^{\mathcal{FD}\to\mathcal{R}}(\pi, M)$.

(iv) If $p\ \overline{t_n}\ \to!\ t$ is a primitive $\mathcal{R}$ constraint π then $M' = M$, $H' = H$, $F' = (F'',\ F)$ and $R' = (\pi,\ R)$, where $F'' = propagations^{\mathcal{R}\to\mathcal{FD}}(\pi, M)$.

where the *propagations* operations given in Tables 3 and 4 take care of the construction of mate constraints via bridges in M for propagation between the $\mathcal{FD}$ and $\mathcal{R}$ stores.

π	$bridges^{\mathcal{FD}\to\mathcal{R}}(\pi, M)$	$propagations^{\mathcal{FD}\to\mathcal{R}}(\pi, M)$
domain $[X_1, \ldots, X_n]\ a\ b$	$\{X_i\#== RX_i \mid 1\leq i\leq n,\ X_i$ has no bridge in M, RX_i new$\}$	$\{a \leq RX_i, RX_i \leq b \mid 1\leq i\leq n\ (X_i\#== RX_i) \in M\}$
belongs $X\ [a_1, \ldots, a_n]$	$\{\ X\#== RX \mid X$ has no bridge in M, RX new$\}$	$\{min(a_1, ..a_n)\leq RX, RX\leq max(a_1, ..a_n)\ 1\leq i\leq n\ (X\#== RX) \in M\}$
$t_1\#{<}t_2$ (analogously $\#{<=}, \#{>}, \#{=>}, \#{=}$)	$\{X_i\#== RX_i \mid 1\leq i\leq 2,\ t_i$ is a variable X_i with no bridge in M, RX_i new$\}$	$\{t_1^{\mathcal{R}} < t_2^{\mathcal{R}} \mid$ For $1\leq i\leq 2$: either t_i is an integer constant n and $t_i^{\mathcal{R}}$ is n, or else t_i is a variable X_i, $(X_i\#== RX_i) \in M$, and $t_i^{\mathcal{R}}$ is $RX_i\}$
$t_1\#{+}t_2\ \to!t_3$ (analogously $\#-$, $\#*$)	$\{X_i\#== RX_i \mid 1\leq i\leq 3,\ t_i$ is a variable X_i with no bridge in M, RX_i new$\}$	$\{t_1^{\mathcal{R}} + t_2^{\mathcal{R}}\ \to!\ t_3^{\mathcal{R}} \mid$ For $1\leq i\leq 3$: $t_i^{\mathcal{R}}$ is determined as in the previous case$\}$

Table 3
Bridge Constraints and Propagations from $\mathcal{FD}$ to $\mathcal{R}$

3.4 Rules for Constraint Solving

The last four rules describe the process of *constraint solving* by means of the application of a constraint solver over the corresponding stores (M, H, F or R). We note that, in order to respect the admissibility conditions of goals and perform an adequate lazy evaluation, we must protect all the produced variables $\chi = pvar(P)$ occurring in the stores from eventual binding caused by the solvers (see [9] for more details). We use the following notations:

- $H \Vdash_{\mathbf{Solver}^{\mathcal{H}},\chi} \exists\overline{Y}.H'$ indicates one of the alternatives computed by the solver.
- $H \Vdash_{\mathbf{Solver}^{\mathcal{H}},\chi} \blacklozenge$ indicates *failure* of the solver (i.e., H is *unsatisfiable*).

Similar notations are used to indicate the behavior of the $\mathcal{FD}$ and $\mathcal{R}$ solvers. The simple behavior of the $\mathcal{M}$ solver is shown explicitly.

MS ***M*****-Solver**

- $\exists\overline{U}.\ P\square C\square\ X\# == u',\ M\square H\square F\square R \Vdash_{\mathbf{MS_1}} \exists\overline{U'}.\ (P\square C\square M\square H\square F\square R)@_F\sigma$

π	$bridges^{\mathcal{R}\rightarrow\mathcal{FD}}(\pi, M)$	$propagations^{\mathcal{R}\rightarrow\mathcal{FD}}(\pi, M)$
$RX{<}RY$	$\emptyset$ (no bridges are created)	$\{X\#{<}Y \mid (X\#{==}RX),(Y\#{==}RY) \in M\}$
$RX{<}a$	$\emptyset$ (no bridges are created)	$\{X\#{<}\lceil a\rceil \mid a{\in}\mathbb{R},(X\#{==}\ RX) \in M\}$
$a{<}RY$	$\emptyset$ (no bridges are created)	$\{\lfloor a\rfloor\#{<}\ Y \mid a{\in}\mathbb{R},(Y\#{==}\ RY) \in M\}$
$RX{<=}\ RY$	$\emptyset$ (no bridges are created)	$\{X\#{<=}Y \mid (X\#{==}RX),(Y\#{==}RY){\in}M\}$
$RX{<=}a$	$\emptyset$ (no bridges are created)	$\{X\#{<=}\lfloor a\rfloor \mid a{\in}\mathbb{R},\ (X\#{==}RX){\in}M\}$
$a <= RY$	$\emptyset$ (no bridges are created)	$\{\lceil a\rceil\#{<=}Y \mid a{\in}\mathbb{R},(Y\#{==}RY){\in}M\}$
$t_1 == t_2$	$\{X\#\ ==\ RX$ | either t_1 is an integer constant and t_2 is a variable RX with no bridges in M (or viceversa) and X is new$\}$	$\{t_1^{\mathcal{FD}} == t_2^{\mathcal{FD}}$ | For 1≤i≤2: either t_i is an integer constant n and $t_i^{\mathcal{R}}$ is n, or else t_i is a variable RX_i, $(X_i\#\ ==\ RX_i) \in M$, and $t_i^{\mathcal{FD}}$ is $X_i\}$
$t_1{+}t_2{\rightarrow}!t_3$ (analogously for $-$, $*$)	$\{X\#{==}\ RX$ | t_3 is a variable RX with no bridge in M, X new, for 1≤i≤2 t_i is either an integer constant or a variable RX_i with bridge $(X_i\#{==}\ RX_i) \in M\}$	$\{t_1^{\mathcal{FD}}\ \#+\ t_2^{\mathcal{FD}}\ \rightarrow!\ t_3^{\mathcal{FD}}$ | For 1≤i≤3: $t_i^{\mathcal{FD}}$ is determined as in the previous case$\}$
$t_1\ /\ t_2\ \rightarrow!\ t_3$	$\emptyset$ (no bridges are created)	$\{t_2^{\mathcal{FD}}\ \#*\ t_3^{\mathcal{FD}}\ \rightarrow!\ t_1^{\mathcal{FD}}$ | For 1≤i≤3 is determined as in the previous case$\}$

Table 4
Bridge Constraints and Propagations from $\mathcal{R}$ to $\mathcal{FD}$

If $X \notin pvar(P)$, $u' \in \mathbb{R}$, $\sigma = \{X \mapsto u\}$ with $u \in \mathbb{Z}$ such that $equiv\ u\ u'$, $\overline{U'} = \overline{U}$ if $X \notin \overline{U}$ and $\overline{U'} = \overline{U} \setminus \{X\}$ otherwise.

- $\exists\overline{U}.\ P\Box C\Box\ u\# == RX,\ M\Box H\Box F\Box R \Vdash_{\mathbf{MS_2}} \exists\overline{U'}.\ (P\Box C\Box M\Box H\Box F\Box R)@_R\sigma$
  If $RX \notin pvar(P)$, $u \in \mathbb{Z}$, $\sigma = \{RX \mapsto u'\}$ with $u' \in \mathbb{R}$ such that $equiv\ u\ u'$, $\overline{U'} = \overline{U}$ if $RX \notin \overline{U}$ and $\overline{U'} = \overline{U} \setminus \{RX\}$ otherwise.
- $\exists\overline{U}.\ P\Box C\Box\ u\# == u',\ M\Box H\Box F\Box R \Vdash_{\mathbf{MS_3}} \exists\overline{U}.\ P\Box C\Box M\Box H\Box F\Box R$
  If $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and $equiv\ u\ u' = true$.
- $\exists\overline{U}.\ P\Box C\Box\ u\# == u',\ M\Box H\Box F\Box R \Vdash_{\mathbf{MS_4}} \blacksquare$
  If $u \in \mathbb{Z}$, $u' \in \mathbb{R}$ and $equiv\ u\ u' = false$.

HS ***H*-Solver**
$\exists\overline{U}.\ P\Box C\Box M\Box H\Box F\Box R \Vdash_{\mathbf{HS}} \exists\overline{Y},\ \overline{U}.\ (P\Box C\Box M\Box H'\Box F\Box R)\sigma'$
If $\chi = pvar(P) \cap var(H)$ and $H \Vdash_{\mathbf{Solver}^{\mathcal{H}},\ \chi} \exists\overline{Y}.\ H'$ with $H' = \Pi' \Box \sigma'$.

FS ***F*-Solver** $\exists\overline{U}.\ P\Box C\Box M\Box H\Box F\Box R \Vdash_{\mathbf{FS}} \exists\overline{Y},\ \overline{U}.\ (P\Box C\Box M\Box H\Box F'\Box R)\sigma'$
If $\chi = pvar(P) \cap var(F)$ and $F \Vdash_{\mathbf{Solver}^{\mathcal{FD}},\ \chi} \exists\overline{Y}.\ F'$ with $F' = \Pi' \Box \sigma'$.

RS R**-Solver** $\exists \overline{U}.\ P \Box C \Box M \Box H \Box F \Box R \Vdash_{\mathbf{RS}} \exists \overline{Y},\ \overline{U}.\ (P \Box C \Box M \Box H \Box F \Box R')\sigma'$
If $\chi = pvar(P) \cap var(R)$ and $R \Vdash_{\mathbf{Solver}^{\mathcal{R}},\ \chi} \exists \overline{Y}.\ R'$ with $R' = \Pi' \Box \sigma'$.

SF Solving Failure
$\exists \overline{U}.\ P \Box C \Box M \Box H \Box F \Box R \Vdash_{\mathbf{SF}} \blacksquare$
If $\chi = pvar(P) \cap var(K)$ and $K \Vdash_{\mathbf{Solver}^{\mathcal{D}},\ \chi} \blacklozenge$, where $\mathcal{D}$ is the domain $\mathcal{H}$, $\mathcal{FD}$ or $\mathcal{R}$ and K is the corresponding constraint store (i.e., H, F or R).

The following example illustrates the process of *flattening* and *propagation*, starting with the real arithmetic constraint $(RX + 2 * RY) * RZ \ \ <= \ 3.5$ and bridges for RX, RY and RZ. At each goal transformation step, we underline the selected subgoal and the applied rule. We use Tables 3 and 4 in order to build new bridges and propagations in the transformation process. In these tables, no bridges are created for $\#/$, because integer division cannot be propagated to real division. The notations $\lceil a \rceil$ (resp. $\lfloor a \rfloor$), stand for the least integer upper bound (resp. the greatest integer lower bound) of $a \in \mathbb{R}$. Constraints $t_1 > t_2$ resp. $t_1 >= t_2$ not occurring in Table 4 are treated as $t_2 < t_1$ resp. $t_2 <= t_1$.

$\Box\ \underline{(RX + 2 * RY) * RZ \ <= \ 3.5} \Box X\# == RX,\ Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash_{\mathbf{FC}}$
$\exists RA.\ \underline{(RX + 2 * RY) * RZ \ \rightarrow \ RA} \Box RA \ <= \ 3.5 \Box X\# == RX,\ Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash_{\mathbf{PC}}$
$\exists RA.\ \Box\ \underline{(RX + 2 * RY) * RZ \ \rightarrow!\ RA},\ RA \ <= \ 3.5 \Box X\# == RX,\ Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash_{\mathbf{FC}}$
$\exists RB, RA.\underline{RX + 2 * RY \rightarrow RB} \Box RB * RZ \rightarrow! RA, RA <= 3.5 \Box X\# == RX, Y\# == RY, Z\# == RZ \Box \Box \Box \Vdash_{\mathbf{PC}}$
$\exists RB, RA.\ \Box\ \underline{RX + 2 * RY \ \rightarrow!\ RB},\ RB * RZ \rightarrow!\ RA,\ RA <= 3.5 \Box X\# == RX,\ Y\# == RY,\ Z\# == RZ$ $\Box \Box \Box \Vdash_{\mathbf{FC}}$
$\exists RC, RB, RA.\ \underline{2 * RY \ \rightarrow \ RC} \Box RX + RC \rightarrow!\ RB,\ RB * RZ \rightarrow!\ RA,\ RA <= 3.5 \Box X\# == RX,$ $Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash_{\mathbf{PC}}$
$\exists RC, RB, RA.\ \Box\ \underline{2 * RY \ \rightarrow!\ RC},\ \underline{RX + RC \ \rightarrow!\ RB},\ \underline{RB * RZ \ \rightarrow!\ RA},\ RA <= 3.5 \Box X\# == RX,$ $Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash^{3}_{\mathbf{SB}}$
$\exists C, B, A, RC, RB, RA.\ \Box\ \underline{2 * RY \ \rightarrow!\ RC},\ \underline{RX + RC \ \rightarrow!\ RB},\ \underline{RB * RZ \ \rightarrow!\ RA},\ \underline{RA \ <= \ 3.5} \Box C\ \# ==$ $RC,\ B\# == RB,\ A\# == RA,\ X\# == RX,\ Y\# == RY,\ Z\# == RZ \Box \Box \Box \Vdash^{4}_{\mathbf{SC}}$
$\exists C, B, A, RC, RB, RA.\ \Box \Box\ C\# == RC,\ B\# == RB,\ A\# == RA,\ X\# == RX,\ Y\# == RY,\ Z\# == RZ$ $\Box \Box\ 2\ \#*\ Y \rightarrow!\ C,\ X\ \#+\ C \rightarrow!\ B,\ B\ \#*\ Z \rightarrow!\ A,\ A\ \# <= 3 \Box 2 * RY \rightarrow!\ RC,\ RX + RC \rightarrow!\ RB,\ RB$ $* RZ \rightarrow!\ RA,\ RA <= 3.5$

4 Implementation and Performance Results

In this section, we present some hints about the implementation of the formal setting presented above, and we test its performance showing the improvements caused by propagation w.r.t. a restricted use of bridges for binding alone.

4.1 Implementation

Our implementation has been developed by adding a store M for bridges, as well as code for implementing bindings and propagations, on top of the existing $\mathcal{TOY}$ system [1]. $\mathcal{TOY}$ has three solvers already for the constraint domains $\mathcal{H}$,

$\mathcal{FD}$ and $\mathcal{R}$, each of them with its corresponding stores. Each predefined function is implemented as a SICStus Prolog predicate that has arguments for: function arguments (as many as its arity), function result, and Herbrand constraint store.

The next example is a simplified code excerpt that shows how the binding mechanism for bridges is implemented. Actually, this is the only code needed for obtaining the performance results shown in Subsection 4.2 for computations without propagation.

```
(1) #==(L, R, true, Cin, ['#=='(HL,HR)|Cout]) :-
(2)   hnf(L, HL, Cin, Cout1), hnf(R, HR, Cout1, Cout),
(3)   freeze(HL, HR is float(HL)), freeze(HR, HL is integer(HR)).
```

This predefined constraint demands its two arguments (`L` and `R`) to be in head normal form (hnf). Therefore, the code line (2) implements the application of the rules **FC** and **PC** (with *true* as t and *equiv* as p). Next, line (3) implements the application of the transformation rule **MS**. The predicate `freeze` suspends the evaluation of its second argument until the first one becomes ground. What we need to reflect in this constraint is to equal two arguments (variables or constants) of different type, i.e., real and integer, so that type casting is needed (`float` and `integer` operations). Binding and matching inherent in **MS** are accomplished by unification. Finally, the transformation rule **SC** (case i) is implemented by adding the flattened bridge to the communication store. The last two arguments of predicate `#==` stand for the input and output stores. For the sake of rapid prototyping, the current implementation mixes Herbrand and communication constraints in one single store, although they should be separated for better performance. In addition, we always add constraints to the communication store (irrespective of groundness) and never drop them; again to be enhanced in a final release.

Implementing propagation requires a modification of existing code for predefined constraints. For example, the code excerpt below shows the implementation of the relational constraint `#>`.

```
(1) #>(L, R, Out, Cin, Cout) :-
(2)  hnf(L, HL, Cin, Cout1), hnf(R, HR, Cout1, Cout),
(3)  searchVarsR(HL,Cout2,Cout3,HLR), searchVarsR(HR,Cout3,Cout,HRR),
(4)  ((Out=true, HL#>HR, {HLR>HRR});(Out=false, HL#<=HR, {HLR<=HRR})).
```

Here, line (2) implements the **FC** and **PC** goal transformation rules; line(3) implements the rule **SB** by adding new needed bridges to the mixed $H + M$ store; and line (4) implements propagation (case (iii) of rule **SC**), sending both the $\mathcal{FD}$ constraint and its mate in $\mathcal{R}$ to the corresponding solvers. Note that, because we allow reification in particular for relational constraints, the complementary cases (`true` and `false` results) correspond to complementary constraints.

4.2 Performance Results

Table 5 compares the timing results for executing the goals in Section 2 for the running example (see Subsection 2.2). The first column indicates the goal, the second and third ones indicate the parameters d and n determining the middle point and the size of the square grid, respectively. The next columns show running times (in milliseconds) in the form (t_B/t_{BP}), where t_B stands for the system using bridges for binding alone and t_{BP} for the system using bridges also for propagation. Values '0' in these columns stand for very small execution times that are displayed as '0' by the system, last columns are headed with a number i which refers to the i-th solution found, and the last column with numbers stands for the time needed to determine that there are no more solutions. In this simple example we see that the finite domain search space has been hugely cut by the propagations from $\mathcal{R}$ to $\mathcal{FD}$. Finite domain solvers are not powerful enough to cut the search space in such an efficient way as simplex methods do for linear real constraints.

Goal	d	n	1	2	3	4	5
1	20000	40000	1828/0	-	-	-	-
	2000000	4000000	179000/0	-	-	-	-
2	20000	40000	1125/0	2172/0	-	-	-
	2000000	4000000	111201/0	215156/0	-	-	-
3	20000	40000	1125/0	1485/0	0/0	1500/0	2203/0
	2000000	4000000	111329/0	147406/0	0/0	147453/0	216156/0

Table 5
Performance Results

5 Conclusions

We have presented a proposal for the cooperation of solvers for the three domains $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$ in Constraint Functional Logic Programming, based on the propagation of mate constraints between the $\mathcal{FD}$ and $\mathcal{R}$ solvers. Our presentation includes both a formal description of cooperative goal solving as an enrichment of existing goal solving calculi [9,2] and a discussion of an effective implementation as an extension of an existing constraint functional logic system, which was already shown to have a reasonable performance [3]. We have obtained encouraging performance results, shown by goal solving examples where the propagation of mate constraints dramatically cuts the search space, thus leading to significant speedups in execution time. Besides the benefits of improving efficiency in a sequential environment, cooperation of solvers even opens the possibility of exploiting emerging technologies such as parallel architectures and grid computing for the parallel execution of different solvers on different processing elements (platforms, processors or cores).

As mentioned in the introduction, the cooperation of constraint solvers has been extensively investigated during the last years [4]. Let us mention at this point just a restricted selection of related work. In his PhD thesis [12] Eric Monfroy proposed BALI (Binding Architecture for Solver integration, see also [13,14]), providing a number of cooperations primitives which can be used to combine various solvers according to different strategies. Monfroy's approach assumes that all the solvers work over a common store, while our present proposal requires communication among different stores. Mircea Marin [10] developed a $CFLP$ scheme that combines Monfroy's approach to solver cooperation with a higher-order lazy narrowing calculus somewhat similar to [9] and the goal solving calculus we have presented in Section 3. In contrast to our proposal, Marin's approach allows for higher-order unification, which leads both to greater expressivity and to less efficient implementations. Moreover, the instance of $CFLP$ implemented by Marin and others [11] is quite different to our work, since it deals with the combination of four solvers over a constraint domain for algebraic symbolic computation. More recently, Petra Hofstedt [6,5] proposed a general approach for the combination of various constraint systems and declarative languages into an integrated system of cooperating solvers. In Hofstedt's proposal, the goal solving procedure of a declarative language is viewed also as a solver, and cooperation of solvers is achieved by two mechanisms: constraint propagation, that submits a constraint belonging to some domain $\mathcal{D}$ to $\mathcal{D}$'s constraint store, say $S_{\mathcal{D}}$; and projection of constraint stores, that consults the contents of a given store $S_{\mathcal{D}}$ and deduces constraints for another domain. Propagation, as used in this paper, is more akin to Hofstedt's projection; while Hofstedt's propagation corresponds to our goal solving rules for placing constraints in stores and invoking constraint solvers. Hofstedt's ideas have been implemented in a meta-solver system called `META-S`, but we are not aware of any performance results.

These and other related works encourage us to continue our investigation, aiming at finding and implementing more elaborated means of communication among solvers, as well as trying their performance experimentally. Future planned work also includes modelling the declarative semantics of cooperation. The implementation described in this paper will be soon available (see `http://toy.sourceforge.net`).

References

[1] Arenas, P., A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo and F. Sáenz-Pérez, $\mathcal{TOY}$. *A Multiparadigm Declarative Language. Version 2.2.2* (2006), R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.

[2] del Vado-Vírseda, R., *Declarative Constraint Programming with Definitional Trees*, in: *Proc. FroCoS'05* (2005), pp. 184–199.

[3] Fernández, A. J., T. Hortalá-González, F. Sáenz-Pérez and R. del Vado-Vírseda, *Constraint Functional Logic Programming over Finite Domains*, Theory and Practice of Logic Programming (2007), in Press.

[4] Granvilliers, L., E. Monfroy and F. Benhamou, *Cooperative Solvers in Constraint Programming: A Short Introduction*, in: *Workshop on Cooperative Solvers in Constraint Programming*, 2001.

[5] Hofstedt, P., "Cooperation and Coordination of Constraint Solvers," Ph.D. thesis, Shaker Verlag. Aachen (2001).

[6] Hofstedt, P. and P. Pepper, *Integration of Declarative and Constraint Programming*, Theory and Practice of Logic Programming (2007), in Press.

[7] Jaffar, J. and M. Maher, *Constraint Logic Programming: a Survey*, Journal of Logic Programming **19&20** (1994), pp. 503–581.

[8] López-Fraguas, F., M. Rodríguez-Artalejo and R. del Vado-Vírseda, *A New Generic Scheme for Functional Logic Programming with Constraints*, Journal of Higher-Order and Symbolic Computation (2007), in Press. Extended version of *Constraint Functional Logic Programming Revisited*, WRLA'04, ENTCS 117, pp. 5-50, 2005.

[9] López-Fraguas, F., M. Rodríguez-Artalejo and R. del Vado-Vírseda, *A Lazy Narrowing Calculus for Declarative Constraint Programming*, in: *Proc. of PPDP'04* (2004), pp. 43–54.

[10] Marin, M., "Functional Logic Programming with Distributed Constraint Solving," Ph.D. thesis, Johannes Kepler Universität Linz (2000).

[11] Marin, M., T. Ida and W. Schreiner, *CFLP: a Mathematica Implementation of a Distributed Constraint Solving System*, in: *Third International Mathematica Symposium (IMS'99)* (1999).

[12] Monfroy, E., "Solver Collaboration for Constraint Logic Programming," Ph.D. thesis (1996).

[13] Monfroy, E., *A Solver Collaboration in BALI*, in: *Proc. of JCSLP'98* (1998), pp. 349–350.

[14] Monfroy, E. and C. Castro, *Basic Components for Constraint Solver Cooperations*, in: *Proc. 18th Annual ACM Symposium on Applied Computing (SAC 2003)* (2003), pp. 367–374.

[15] SICStus Prolog (2006), `http://www.sics.se/isl/sicstus`.