Cooperation of Constraint Domains in the TOY System

Sonia Estévez-Martín
Dpto. de Sist. Inform. y Comp.
Univ. Complutense de Madrid
s.estevez@sip.ucm.es

Antonio J. Fernández
Dpto. de Leng. y C. Comp.
Univ. de Málaga
afdez@lcc.uma.es

Teresa Hortalá-González
Dpto. de Sist. Inform. y Comp.
Univ. Complutense de Madrid
teresa@sip.ucm.es

Mario Rodríguez-Artalejo
Dpto. de Sist. Inform. y Comp.
Univ. Complutense de Madrid
mario@sip.ucm.es

Fernando Sáenz-Pérez
Dpto. de Ing. Soft. e Int. Art.
Univ. Complutense de Madrid
fernan@sip.ucm.es

Rafael del Vado Vírseda
Dpto. de Sist. Inform. y Comp.
Univ. Complutense de Madrid
rdelvado@sip.ucm.es

Abstract

This paper presents a computational model for the cooperation of constraint domains, based on a generic Constraint Functional Logic Programming (CFLP) Scheme and designed to support declarative programming with functions, predicates and the cooperation of different constraint domains equipped with their respective solvers. We have developed an implementation in the CFLP system TOY, supporting an instance of the scheme which enables the cooperation of symbolic Herbrand constraints, finite domain integer constraints, and real arithmetic constraints. We provide a theoretical result and an analysis of benchmarks showing a good performance with respect to the closest related approach we are aware of.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.6 [Programming Techniques]: Logic Programming; D.3.2 [Programming Languages]: Language Classifications—Multiparadigm languages; D.3.2 [Programming Languages]: Language Classifications—Constraint and logic languages; D.3.3 [Programming Languages]: Language Constructs and Features—Constraints

General Terms Algorithms, Languages, Performance, Theory.

Keywords Functional and Logic Programming, Constraint Domains and Solvers, Domain Cooperation.

1. Introduction

Constraint Programming relies on constraint solving as a powerful mechanism for practical applications. The well-known CLP scheme serves as foundation for a powerful and practical framework for Constraint Logic Programming [11], inheriting the clean semantics and declarative style of Logic Programming. The combination of CLP with Functional Programming has given rise to various so-called CFLP schemes, aiming at a very expressive combination of the constraint, logical and functional programming paradigms. This paper tackles foundational and practical issues concerning the efficient use of constraints in CFLP languages and systems. Both the CLP and the CFLP schemes must be instantiated by a constraint domain D which provides specific data values and solvers. Useful constraint domains include the Herbrand domain H for equality and disequality constraints over symbolic terms; the domain R for arithmetic constraints over real numbers; and the domain FD for finite domain constraints over integer values.

Practical applications, however, often involve more than one ‘pure’ domain, and sometimes problem solutions have to be artificially adapted to fit a particular choice of domain and solver. Combining decision procedures for theories is a well-known problem, thoroughly investigated since the seminal paper of Nelson and Oppen [17]. In constraint programming, however, the emphasis is placed in computing answers by the interaction of constraint solvers with user given programs, rather than in deciding satisfiability of formulas. The cooperative combination of constraint domains and solvers has evolved during the last decade as a relevant research issue that is raising an increasing interest in the CLP community. Here we mention [2, 8, 12, 16, 10] as a limited selection of references illustrating various approaches to the problem. An important idea emerging from the research in this area is that of ‘hybrid’ constraint domain, built as a combination of simpler ‘pure’ domains and designed to support the cooperation of its components, so that more declarative and efficient solutions for practical problems can be promoted.

The aim of this paper is to present a computational model for the cooperation of constraint domains in the CFLP context, along with an implementation for a particular case of practical importance. We rely on the CFLP scheme [15] and the goal solving calculus [14], enriched with cooperation mechanisms inspired in our previous papers [4, 5], where so-called bridges were introduced as a key tool for communicating constraints between different domains. Bridges are constraints of the form $X \#==_{d_i,d_j} Y$ which relate the values of two variables $X : d_i$, $Y : d_j$ of different base types, requiring them to be equivalent. For instance, $X \#==_{\text{int,real}} Y$ (abbreviated as $X \#== Y$ in the rest of the paper) constrains the real variable $Y : \text{real}$ to take an integral real value equivalent to that of the integer variable $X : \text{int}$. Note that the two types int and real are kept distinct and their respective values are not confused.
We use instances of the CFLP scheme [15] given by coordination domains, built as a `hybrid' combination of various `pure' domains intended to cooperate, plus a mediatorial domain supplying bridge constraints. Different constraint stores are assigned to the various domains and solvers. In addition, a special mediatorial store keeps the bridge constraints which arise during the computation. A bridge constraint \( X \equiv Y \) available in the mediatorial store can be used to project constraints involving the variable \( X \) into constraints involving the variable \( Y \), or vice versa. For instance, the constraint \( \mathcal{R} \) constraint \( RX \geq d - 0.5 \) (based on the inequality primitive \( - \) – `less or equal' – for the type real) can be projected into the \( FD \) constraint \( X \equiv 3 \) (based on the inequality primitive \( \equiv \) – `less or equal' – for the type int) in case that the bridge \( X \equiv Y \) is available. Projected constraints are submitted to their respective stores, with the aim of improving the performance of the corresponding solvers. We have borrowed the idea of constraint projection from [10], adapting it to our CFLP scheme and adding bridge constraints with the aim of making projections more flexible and compatible with type discipline. In order to illustrate the use of projections, assume a given even integer number \( n = 2d \) and consider the problem of solving the following constraint system \( E \):

\[
X \equiv RX, \ Y \equiv RY,
\]

\[
RY \geq d - 0.5, \ RX - RX \leq -0.5, \ RX + RX \geq n + 0.5,
\]

domain \([X, Y] 0 \text{ n} \), labeling \([X, Y]\).

This aims at computing the intersection of a triangular region in the continuous plane, with vertices at the points \((d - 1, d - 0.5), (d, d - 0.5), (d + 1, d - 0.5)\) described by the inequalities \( RX \geq d - 0.5, \ RX - RX \leq 0.5, \ RX + RX \geq n + 0.5\) with a \( n \times n \) square grid of points in the discrete plane, centered at the point \((d, d)\) described by the finite domain constraint domain \([X, Y] 0 \text{ n}\). The last constraint labeling \([X, Y]\) forces the enumeration of single solutions. When solving this system the two available bridges can be used to project the \( \mathcal{R} \) constraints \( RY \geq d - 0.5, \ RX - RX \leq 0.5, \ RX + RX \geq n + 0.5 \) into the \( FD \) constraints \( Y \equiv d, \ Y - X \equiv 0, \ Y + X \equiv 0 \). Since \( n = 2d \), the only possible solution is \( X = Y = d \). Therefore, the \( FD \) solver drastically prunes the domains of \( X \) and \( Y \) to the singleton set \([d]\), and solving the labeling constraint leads to the unique solution with no effort. For a big value of \( n = 2d \), the use of projections greatly enhances the performance of the computation, as shown by the experimental results given in [4] and Section 4 below.

We have implemented our cooperative CFLP computation model for a particular instance of practical interest, supporting the cooperation of the domains \( H, FD \) and \( R \). The implementation has been developed in the TOY system [1], which is implemented in SICStus Prolog [18]. In order to provide some evidence for the performance of our implementation, we have worked out a set of benchmarks tailored to test the efficiency of our system in comparison with the closest related system we are aware of, namely the META-S tool [6] which implements the framework for solver cooperation described in [10]. The experimental results we have obtained are quite encouraging.

The rest of the paper is structured as follows: Section 2 recalls the essentials of the CFLP scheme and presents the construction of coordination domains \( C \) from `pure' domains given as components. All the subsequent sections deal with a particular choice of \( C \) tailored to the cooperation of \( H, R \) and \( FD \). Section 3 presents our computational model for cooperative goal solving in the instance CFLP(\( C \)) of CFLP. Section 4 analyzes a series of benchmarks worked out to test the performance of our implementation in comparison with META-S. Section 5 summarizes the contributions of the paper and points to some lines for future work.

2. Cooperation of Constraint Domains

In this section we recall the essentials of the CFLP scheme, extending [15] with a polymorphic type discipline in the style of [7]. Then, we present a mathematical construction of coordination domains and the particular coordination domain \( D \) tailored to the cooperation of \( H, FD \) and \( R \).

2.1 Signatures and Expressions

We assume a universal signature \( \Omega = \langle TC, BT, DC, DF, PF \rangle \) consisting of pairwise disjoint sets of Type Constructors

\[
TC = \bigcup_{n \in \mathbb{N}} TC^n, \ Base \ Types \ BT, \ Data \ Constructors \ DC = \bigcup_{n \in \mathbb{N}} DC^n, \ Defined \ Function \ Symbols \ DF = \bigcup_{n \in \mathbb{N}} DF^n \end{equation}

and Primitve Function Symbols \( PF = \bigcup_{n \in \mathbb{N}} PF^n \). Types are built from type constructors and type variables in the usual way, and each symbol in \( DC \cup DF \cup PF \) comes with an attached principal type: see e.g. [7] for details. In particular, the polymorphic primitive \( :: = \ A \rightarrow A \rightarrow \text{bool} \) belonging to \( PF \) can be used in infix notation to write strict equality and disequality constraints.

We consider domain specific signatures \( \Sigma = \langle TC, BT, DC, DF, SPF \rangle \), with \( \Sigma \subseteq \mathcal{B} \) and \( \Sigma \subseteq \mathcal{F} \). Given any \( \Sigma \) and any \( \Sigma \)-indexed family \( \mathcal{B} = \{ B_{\eta} | \exists \mathcal{B} \} \) of non-empty sets \( B_\eta \) of base values \( \eta \) of type \( \Sigma \), we define \( \Sigma \)-expressions \( e \in \mathcal{E}_{\Sigma} (\mathcal{B}) \) over \( \Sigma \) with syntax \( e ::= X \mid \bot \mid u \mid h \mid e_1 e_2 \), where \( X \) is a variable, \( u \in B \), \( h \in DC \cup DF \cup SPF \), and \( e_1 e_2 \) stands for the application of \( e \) as function to \( e_1 \) as argument. We abbreviate \( \ldots (\ldots (\ldots e_1 e_2 \ldots \ldots e_n) \ldots \ldots) \) as \( e_n \). Expressions without occurrences of the undefined value \( \bot \) in \( \mathcal{B} \) are called total. \( \Sigma \)-patterns \( p \in \mathcal{P}_{\Sigma}^{\mathcal{B}} (\mathcal{B}) \) over \( \mathcal{B} \) are particular expressions representing data values. Its syntax is \( t ::= X \mid \bot \mid u \mid e_1 \tau_1 \mid f_1 \tau_1 \mid p \tau_1 \), where \( X \) is a variable, \( u \in B \), \( h \in DC \), \( f \in DF \) for some \( n \geq m \), \( f \in DF \) for some \( n > m \), and \( p \in SPF \) for some \( n > m \). The set of all ground patterns \( (\Sigma) \), (variable-free patterns) over \( \mathcal{B} \) is called \( G\Sigma \mathcal{P} \mathcal{B} (\mathcal{B}) \), viewed as the universe of values \( \mathcal{B} \) and written also as \( \mathcal{U} \).

2.2 Constraint Domains

Constraint domains of signature \( \Sigma \) are structures \( D = (B_\Sigma, \{ p_\Sigma \}_{p \in SPF}) \), where \( B_\Sigma = \{ B_\Sigma^p | p \in SPF \} \) is a \( \Sigma \)-indexed family of sets of base values and the interpretation \( e \) of each primitive function symbol \( p \) of \( \Sigma \) is a function \( p_\Sigma \) of \( e \) over \( B_\Sigma \) satisfying the following properties:

- For any \( \Sigma \)-patterns \( p \) of \( \Sigma \) there is a \( \Sigma \)-expression \( e \) such that \( p_\Sigma (e) \) is defined.
- For any \( \Sigma \)-expression \( e \), \( p_\Sigma (e) \) is total.
- \( p_\Sigma \) is a \( \Sigma \)-function on \( \mathcal{E}_{\Sigma} (\mathcal{B}) \).
- \( p_\Sigma \) is monotonic.
- \( p_\Sigma \) is continuous.

As atomic constraints over \( D \) we use \( \xi \) (standing for truth), \( \xi \) (standing for falsity), and \( p_\Sigma \rightarrow t \) with \( p \in SPF \), intended to mean that \( p \xi \) can return a value equal to the total pattern \( t \). Other constraints are built from atomic ones by means of conjunction and existential quantification. A constraint \( \pi \) is called primitive iff all the atomic parts of \( \pi \) have one of the three forms \( \xi \), \( p_\Sigma \rightarrow t \), where \( p_\Sigma \rightarrow t \) are patterns. We write \( PCon \) (resp., \( ACon \)) for the set of all primitive (resp., atomic primitive) constraints over \( D \).

For any domain \( D \), the set \( Val_D \) of valuations over \( D \) consists of all the substitutions \( \eta \) mapping variables into ground patterns. For any \( \pi \in PCon_D \) we define \( Sol_D (\pi) \subseteq Val_D \) as the set of all \( \eta \in Val_D \) which satisfy \( \pi \), called solutions of \( \pi \). For any \( \Pi \subseteq PCon_D \) we define \( Sol_D (\Pi) = \{ \eta \in Sol_D (\pi) | \forall \pi \in \Pi \} \) as the set of all solutions of \( \Pi \) that are solutions of all \( \pi \in \Pi \). For any \( \pi \in ACon_D \) we postulate an effective way of computing a set of variables \( od\varpi (\pi) \), called the set of variables
obviously demanded by \( \pi \), such that \( \eta(X) \neq \perp \) for all \( X \in odvar(\pi) \) and all \( \eta \in Sol_\Pi(\Pi) \). This can be easily done for the three domains \( \mathcal{H}, \mathcal{F}D \) and \( \mathcal{R} \). For any finite \( \Pi \subseteq A PC_\Pi \) we define \( odvar(\Pi) = def \bigcup_{\eta \in \Pi} odvar(\pi) \) and \( var(\Pi) = def \bigcup_{\eta \in \Pi} var(\Pi) \cup odvar(\Pi) \). Variables in the set \( \eta(\Pi) \) are called critical.

The computation model discussed in the next section uses obviously demanded variables to ensure a lazy evaluation of function calls, as well as constraint stores of the form \( S = \Pi \square \sigma \), where \( \Pi \subseteq A PC_\Pi \) and \( \sigma \) is an idempotent substitution such that \( dom(\sigma) \cap var(\Pi) = \emptyset \). The set of solutions \( Sol_\Pi(\Pi) \) of a given store has a natural definition.

### 2.3 Pure Domains vs Coordination Domains

Constraint domains such as \( \mathcal{H}, \mathcal{F}D \) and \( \mathcal{R} \) are pure in the sense that they are not designed as a combination of simpler domains. A solver for a given pure domain \( D \) is a device which can apply any finite set \( \Pi \subseteq A PC_\Pi \) to an equivalent simpler form. We formalize a solver as a function \( solve() \) which can be applied to a pair \((\Pi, \lambda)\), (with finite \( \Pi \subseteq A PC_\Pi \), and \( \lambda \subseteq \eta(\Pi) \)) and returns a finite disjunction \( \bigvee_{j=1}^{k} \exists \Psi_j(\Pi', \square \sigma_j) \) of existentially quantified constraint stores, written just as a failure symbol in the case \( k = 0 \), and fulfilling three key properties:

(a) **discrimination w.r.t. critical variables** requires that every \( 1 \leq j \leq k \) verifies either \( \lambda \cap odvar(\Psi_j) \neq \emptyset \) or else \( \lambda \cap var(\Psi_j) = \emptyset \);

(b) **soundness** requires that \( Sol_\Pi(\Pi) \supseteq \bigcup_{j=1}^{k} Sol_\Pi(\Psi_j(\Pi', \square \sigma_j)) \);

(c) **completeness w.r.t. well-typed solutions** requires that \( \mathcal{W}T_\pi(\Pi) \subseteq \bigcup_{j=1}^{k} \mathcal{W}T_\pi(\Psi_j(\Pi', \square \sigma_j)) \).

In practice, condition c) may hold only for some choices of the constraint set \( \Pi \) to be solved.

In order to formalize the cooperation of several pure domains, we build coordination domains using an amalgamated sum construction relying on a joinability condition. Two given constraint domains \( D_1 \) and \( D_2 \) with specific signatures \( \Sigma_i = (\mathcal{S}C, SBT_i, DC, DF, SPF_i) \) (i \( \in \{1, 2\} \)) are called joinable if \( \mathcal{S}PF_i \cap \mathcal{S}PF_2 \subseteq \{\emptyset\} \), and for every common base type \( d \in SBT_1 \cap SBT_2 \), one has \( B_{d,1} = B_{d,2} \). The amalgamated sum \( S = D_1 \oplus D_2 \) of two joinable domains \( D_1 \) and \( D_2 \) is then defined as a new domain with signature \( \Sigma = (\mathcal{S}C, SBT \cup SBT_2, DC, DF, SBT_1 \cup SBT_2, SPF_1 \cup SPF_2) \), constructed as follows: \( B_{d,1}^S = B_{d,1} \) for all \( i = 1, 2 \), and \( d \in SBT_1 \) (no conflict will arise for those \( d \in SBT_1 \cap SBT_2 \), because of joinability) and \( p^S \) is defined as the least extension of \( p^S \) which fulfills the properties required for the interpretation of any primitive operation in [15], for all \( p \in SPF_i \), of arity \( n \) other than \( = \) (the interpretation of \( = \) in \( S \) behaves as strict equality over \( S \)’s universe). The amalgamated sum \( D_1 \oplus \cdots \oplus D_n \) of \( n \) pairwise joinable domains \( D_i \) (\( 1 \leq i \leq n \)) with specific signatures \( \Sigma_i = (\mathcal{S}C, SBT_i, DC, DF, SPF_i) \) (\( 1 \leq i \leq n \)) can be defined analogously.

### 3. Cooperative Programming and Goal Solving

In this section we present programs and goals, a selection of the transformation rules more relevant for cooperative goal solving, and a semantic result. A full presentation of cooperative goal solving should borrow from [5] some additional goal solving rules, for tasks such as flattening constraints that involve nested functions, and evaluating calls to program defined functions by means of lazy narrowing.

#### 3.1 Programs and Goals

CFLP(C)-programs are sets of program rules. A program rule for \( f \in DF^n \) has the form \( f_{\eta_0} \leftarrow \Delta \), where \( \eta_0 \) is a linear sequence of patterns, \( \eta_0 \) is an expression and \( \Delta \) is a finite conjunction \( \delta_1, \ldots, \delta_m \) of atomic constraints \( \delta_i \) over \( G \). Goals for such programs have the form \( G \in \mathcal{U} \). Programs for the set \( \mathcal{U} \) of all variables occurring in \( \delta_1, \ldots, \delta_m \).

- \( C \) is the so-called constraint pool, a finite set of constraints to be solved.

- \( M = \Pi \mathcal{M} \square \sigma_M \) is the mediator store, including a finite set of atomic primitive constraints \( \Pi \mathcal{M} \subseteq \mathcal{A}PC_\mathcal{M} \) and a substitution \( \sigma_M \). We will write \( \mathcal{B}_M \subseteq \Pi \mathcal{M} \) for the set of all \( \pi \in \Pi \mathcal{M} \) that are bridges.

- \( H \) (resp., \( F \) and \( R \)) is the \( \mathcal{H} \)-store (resp., \( \mathcal{F}D \)-store and \( \mathcal{R} \)-store), including a finite set of atomic primitive constraints over the corresponding domain and a substitution.

A C-program allowing to solve goals related to the constraint system \( E \) discussed in Section 1 can be found in [4]. The set of variables obviously demanded by a goal \( G \) is defined as the least subset \( odvar(G) \subseteq var(G) \) including all the variables obviously demanded by the stores in \( G \) and all the variables \( X \) in some production \( (X_{\eta_k} \rightarrow t) \in P \) such that either \( t \) is not a variable or else \( k > 0 \) and \( t \in odvar(G) \).

Programs and goals are expected to be well-typed w.r.t. the principal types of the symbols belonging to the underlying signature. Some special goals of interest are: initial goals just consisting of a constraint pool \( C \), empty stores, and no existential variables solved
goals with empty pool and stores in solved form; and the inconsistent goal ■. Solved goals are used as computed answers, while ■ is used to indicate failure.

The set of solutions $\text{Sol}_P(G)$ of goal $G$ w.r.t. program $P$ consists of all $\pi \in V \ast S$ such that $G\pi$ can be derived from $P$ in the Constrained Rewriting Logic formalized by the rules presented in [15]. The set of well-typed solutions of $G$ is defined as $WT\text{Sol}_P(G) = \{ \pi \in \text{Sol}_P(G) | G\pi \text{ is well-typed} \}$. These notions will be used in the next subsection.

### 3.2 Rules for Domain Cooperation and Constraint Solving

Given a goal $G$ whose constraint pool $C$ includes a primitive atomic constraint $\pi \in PC_{\text{Con}_R}$, our computational model can perform the following transformations:

(a) compute new bridges to add to $M$ by means of a bridge generation operation; $\text{bridges}^{R \rightarrow FD} \rightarrow (\pi, B_M)$.

(b) compute projected $R$-constraints to be added to $R$, by means of a projection operation $\text{proj}^{FD \rightarrow R} (\pi, B_M)$.

(c) submit $\pi$ into the store $F$.

For a primitive atomic constraint $\pi \in PC_{\text{Con}_R}$ occurring in $C$, similar transformations based on two operations $\text{bridges}^{R \rightarrow FD}$ and $\text{proj}^{R \rightarrow FD}$ are also possible. Natural and constructive definitions for the bridges and projections operations can be found in [4, 5]. These operations can be proved safe in the sense that they do not miss any solution. The goal transformations just discussed are formalized by the rules $\text{SB}$, $\text{PP}$ and $\text{SC}$ in the upper half of Figure 1. Pragmatically, they should be applied in this order.

Consider solving the goal $G_E$:

$$\Box \mathbb{X} = \mathbb{R}_X, \mathbb{Y} = \mathbb{R}_Y, \mathbb{R} = \var{0, 0.5}; \mathbb{R}_X < 0.5, \mathbb{R}_Y + \mathbb{R}_X < 0.5, \mathbb{R}_Y + \mathbb{R}_X < 0.5; \mathbb{D} \not= \Box, \mathbb{L} \not= \Box, \mathbb{M} \not= \Box$$

Corresponding to the constraint system $E$ discussed in Section 1, First, the two bridges $\mathbb{X} = \mathbb{R}_X, \mathbb{Y} = \mathbb{R}_Y$ are placed into $M$ by applying $\text{SC}$. Due to the nested occurrence of $\mathbb{R} \not= 0.5$ in the pool, this constraint is replaced by $\mathbb{R}_X \not= 0.5$ with a new existential variable $\mathbb{R}_Z$ using flattening transformations. At this point, $\text{SB}$ can be applied, and $\text{bridges}^{R \rightarrow FD} (\mathbb{R}_X \not= 0.5, \mathbb{R}_Z \not= 0.5)$ returns a new bridge $\mathbb{Z} = \mathbb{R}_Z$ which is added to $M$. Now, $\text{PP}$ allows to compute $\text{proj}^{R \rightarrow FD} (\mathbb{R}_Z \not= 0.5, \mathbb{L}_M)$ which returns $\mathbb{Z} \not= 0$. Similar transformations are possible for the other constraints remaining in the pool. In general, $\text{PP}$ submits the projected constraints to their corresponding stores, while constraints in the pool that are not useful for computing additional bridges or projections can be also moved to their stores by means of $\text{SC}$.

The lower half of Figure 1 presents the goal transformations resulting from the invocation of constraint solvers. More precisely, $\text{CS}$ specifies a computation of alternatives and $\text{SF}$ specifies the detection of failure by a solver. Both of these rules require the condition $\text{var}^\pi(P) \cap \text{odvar}_{\text{DF}}(\Pi_S) = \emptyset$, meaning that a constraint solver can be invoked only if no produced variable of the goal is obviously demanded by the constraints placed in the affected store $S$. If this were not the case, constraint solving should be preceded by other goal transformations applied to those productions $e \rightarrow t$ in the goal such that $e$ includes obviously demanded variables. The set of critical variables for the solver invocation $\text{solve}^\pi(\Pi_S, X)$ is chosen as $X = \text{def} \text{var}^\pi(P) \cap \text{var}(\Pi_S)$. Except in the case that $S$ is the Herbrand store $H$, the condition $\text{var}^\pi(P) \cap \text{odvar}_{\text{DF}}(\Pi_S) = \emptyset$ is equivalent to $\text{var}^\pi(P) \cap \text{var}(\Pi_S) = \emptyset$, and therefore $X = \emptyset$.

In addition to the rules in Figure 1, our cooperative goal solving calculus includes other goal transformation rules similar to those in [5]. We write $G \vdash_P S$ to indicate that the goal $G$ can be transformed into the solved goal $S$ in finitely many goal transformation steps. Using proof techniques similar to those in [14, 7] we prove the following theorem, which is a substantial improvement of the full soundness result given in [5]. The technical notions ‘free occurrence of a higher-order logic variable’ and ‘opaque decomposition step’ are used in the statement of the theorem. Roughly, ‘free occurrence higher-order logic variable’ refers to a variable being used as a function but not given as a parameter, while ‘opaque decomposition step’ refers to a special kind of goal solving step that can convert a well-typed goal into an ill-typed one. The reader is referred to [7] for precise explanations.

**Theorem 1** (Soundness and Completeness).

Let $G$ be a goal for a CFLP($G$)-program $P$. Then:

(a) **Soundness:** For any solved goal $S$ such that $G \vdash_P S$, one has $\text{Sol}_c(S) \subseteq \text{Sol}_P(G)$.

(b) **Completeness:** Assume that neither $P$ nor $G$ has free occurrences of higher-order variables. Let $\mu \in WT\text{Sol}_P(G)$ be arbitrarily fixed. Then, unless prevented by an incomplete solver invocation or an opaque decomposition step, one can build a computation $G \vdash_P S$ ending with a solved goal $S$ such that $\mu \in WT\text{Sol}_c(S)$ holds for some $\mu = [\text{var}(G)]$.

The proof of Theorem 1 relies on the following theorem, which provides soundness and completeness results for the one-step transformation of a given goal.

**Theorem 2** (Sound. and Compl. for One-Step).

Assume a given CFLP($G$)-program $P$ and an admissible goal $G$ for $P$ which is not in solved form. Choose any rule $\text{RL}$ applicable to $G$ and select a part of $G$ suitable for applying $\text{RL}$. Then there exist finitely many admissible goals $G_k (1 \leq k \leq l)$ such that $G \vdash_{\text{RL} \cup \text{PP} \cup \text{SC}} G_k$, and moreover:

1. **Soundness:** $\text{Sol}_P(G) \supseteq \bigcup_{k=1}^l \text{Sol}_P(G_k)$.

2. **Completeness:** $\bigcap_{k=1}^l \text{WTSol}_P(G_k)$ holds under two additional assumptions:

   (a) Neither $P$ nor $G$ have free occurrences of higher-order variables.

   (b) The application of $\text{RL}$ to $G$ does not involve an incomplete solver invocation or an opaque decomposition step.

**Proof of Theorem 2.**

We prove that $\text{Sol}_P(G) \supseteq \bigcup_{k=1}^l \text{Sol}_P(G_k)$ and $\text{WTSol}_P(G) \subseteq \bigcap_{k=1}^l \text{WTSol}_P(G_k)$ under the assumptions of the theorem for the goal transformation rules $\text{SB}, \text{PP}, \text{SC}, \text{CS}$, and $\text{SF}$ in Figure 1, and the condition that $G_k (1 \leq k \leq l)$ are the $l$ admissible goals such that $G \vdash_{\text{RL} \cup \text{PP} \cup \text{SC}} G_k$. In each case, we assume that $G$ and $G_k (1 \leq k \leq l)$ are exactly as they appear in the presentation of each corresponding rule in Figure 1, therefore $k = 1$ and let us rename $G_1$ as $G'$. A full proof of the theorem should be completed with similar reasonings for other goal solving transformations similar to those presented in [5], that are not displayed in Figure 1 but belong to our cooperative goal solving calculus.

**Rule SB, Set Bridges:**

Assume that $\mu \in \text{Sol}_P(G')$. There exists $\mu' = \prod_{\mu} \prod_{\mu} \mu$ such that $\mu'$ is a solution for the result of dropping the existential prefix $\exists \mathbb{V}. \mathbb{C}$ of $G'$. In particular, $\mu' \in \text{Sol}_c(M')$, and since $M' = M \cup M'$, we have $\mu' \in \text{Sol}_c(M \cup M')$. Therefore, $\mu' \in \text{Sol}_c(M)$.

Since $\mathbb{V}$ are fresh local variables not occurring in $G$, and the logical conditions in $G$ under the existential prefix $\exists \mathbb{M}$ are the same as those in $G'$, we conclude that $\mu \in \text{Sol}_P(G)$.

It proves $\text{Sol}_P(G) \supseteq \text{Sol}_P(G')$. 
Figure 1. Rules for Domain Cooperation and Constraint Solving

Assume now that $\mu \in \text{WTSol}_F(G)$. There exists $\mu' = \mathcal{U} \mu$ such that $\mu'$ is a solution for the result of dropping the existential prefix $\mathcal{U}$ of $G$. In particular, $\mu' \in \text{WTSol}_F(\pi)$, and $\mu' \in \text{WTSol}_F(M)$, therefore $\mu' \in \text{WTSol}_F(\pi \land M)$.

Let us see that $\mu' \in \text{WTSol}_F(M')$.

First, we observe that one of the following cases can be applied:

(i) If $\pi$ is a $\mathcal{F}D$ constraint then $\mathcal{U}$ returns a finite set $B''$ of new bridge constraints involving new variables $V$, and following the correctness condition hold in the Subsection Bridges and projections for Cooperative Goal Solving of [5]: $\text{WTSol}_F(\pi \land M) \subseteq \text{WTSol}_F(\mathcal{U}(\pi \land M \land B''))$. Then $\mu' \in \text{WTSol}_F(\mathcal{U}(\pi \land M \land B''))$. By definition, there exists $\mu'' = \mathcal{V} \mu'$ such that $\mu'' \in \text{WTSol}_F(\pi \land M \land B'')$. Therefore, $\mu'' \in \text{WTSol}_F(\pi \land M')$ (because $M' \equiv B'' \land M$) and $\mu'' \in \text{WTSol}_F(M')$.

Since the variables in $V$ are fresh local variables and do not occur in $G$, we can conclude that $\mu''$ is a solution of every logical condition in $G'$ under the existential prefix $\mathcal{U}$.

Therefore, we can conclude that $\mu' \in \text{WTSol}_F(G')$.

(ii) Analogous to (i).

It proves $\text{WTSol}_F(G) \subseteq \text{WTSol}_F(G')$.

Rule PP. Propagate Projections:

Assume that $\mu \in \text{Sol}_F(G')$. There exists $\mu' = \mathcal{U} \mu$ such that $\mu'$ is a solution for the result of dropping the existential prefix $\mathcal{U}$ of $G'$, then $\mu' \in \text{Sol}_F(F')$ and $\mu' \in \text{Sol}_F(R')$.

One of the following cases can be applied:

(i) If $\pi$ is a $\mathcal{F}D$ constraint then $F' = F$ and trivially $\mu' \in \text{Sol}_F(F') = \text{Sol}_F(F)$. Furthermore, $\mu' \in \text{Sol}_F(R') = \text{Sol}_F(G' \land R)$ and then $\mu' \in \text{Sol}_F(R)$.

Since $V$ are fresh local variables not occurring in $G$ and the logical conditions in $G$ under the existential prefix $\mathcal{U}$ are the same as those occurring in $G'$, we conclude that $\mu \in \text{Sol}_F(G)$.

(ii) Analogous to (i).

It proves $\text{Sol}_F(G) \supseteq \text{Sol}_F(G')$.
(i) If \( \pi \) is a FD constraint then \( \text{proj}^\text{FD}\to\text{R} (\pi, B_M) \) returns a conjunction of new real constraints \( \Sigma' \) involving new variables \( \Sigma' \). Following the correctness condition hold in the Subsection Bridges and projections for Cooperative Goal Solving of [5]:

\[
\text{WTSol}(\pi \land M) \subseteq \text{WTSol}(\Sigma' \land \Pi' \land M).
\]

By definition, there exists \( \mu'' = \pi \mu' \) such that \( \mu'' \in \text{WTSol}(\pi \land \Pi' \land M) \).

Since \( \Sigma' \) are new variables not occurring in \( G \) and the logical conditions in \( G \) under the existential prefix \( \exists \Sigma' \) are the same, \( \mu'' \in \text{WTSol}(\Sigma' \land \Pi' \land M) \). In particular \( \mu'' \in \text{WTSol}(\Pi') \).

Therefore \( \mu'' \in \text{WTSol}(\Pi' \land R) \), or equivalently, \( \mu'' \in \text{WTSol}(\Pi') \).

Moreover, since \( F = 0 \) then \( \mu'' \in \text{WTSol}(F') \).

Since \( \mu'' = \exists \Sigma' \Pi' \mu' \) then \( \mu'' = \exists \Sigma' \Pi' \mu \), and therefore \( \mu \in \text{WTSol}(G') \).

(ii) Analogous to (i).

It proves \( \text{WTSol}(G) \subseteq \text{WTSol}(G') \).

**Rule SC, Submit Constraints:**

We consider the particular case in which the constraint domain is the mediator domain (the other domains can be proved in a similar way). Therefore, \( S \) is the mediator store \( M, S' \) is \( M' \), 
\( M' = \pi \land M, H' = H, F' = F, \) and \( R' = R \).

We assume that \( \mu \in \text{Sol}(G) \). There exists \( \mu' = \exists \mu \) such that \( \mu' \in \text{Sol}(G) \), and therefore \( \mu'' \in \text{Sol}(M') \) since \( M' = \pi \land M, \mu'' \in \text{Sol}(\pi \land M) \). Therefore, \( \mu'' \in \text{Sol}(\pi) \) and \( \mu'' \in \text{Sol}(M) \).

Since variables and logical conditions in \( G' \) and \( G' \) are the same, we conclude that \( \mu \in \text{Sol}(G) \).

It proves \( \text{Sol}(G) \subseteq \text{Sol}(G') \).

Assume now that \( \mu \in \text{WTSol}(G) \). There exists \( \mu' = \exists \mu \) such that \( \mu' \in \text{WTSol}(\pi) \) and \( \mu'' \in \text{WTSol}(M) \).

We deduce that \( \mu'' \in \text{WTSol}(M') \) (because \( M' = \pi \land M \)).

Since variables and logical conditions in \( G' \) and \( G' \) are the same, we conclude that \( \mu \in \text{WTSol}(G') \).

It proves \( \text{WTSol}(G) \subseteq \text{WTSol}(G') \).

**Rule CS, Constraint Solver:**

As in the previous rule we only consider a particular case in which \( S \) is the finite domain store \( F \). The proofs for \( M, H, \) and \( R \) stores are quite similar. We note that \( R' = R = M' = M, H' = H, \) \( F' = F, \Pi' = \Pi', \sigma' = \sigma, \) and \( Y' = Y' \).

We consider \( \mu \in \text{Sol}(G) \). There exists \( \mu' = \exists \Pi' \sigma' \mu \) such that \( \mu' \in \text{Sol}(P \land C \land M \land H \land \Pi' \land \sigma) \) \( \sigma' \).

Therefore, \( \sigma' \mu \in \text{Sol}(P \land C \land M \land H \land \Pi' \land \sigma) \).

We know now that \( \mu' \in \text{Sol}(\Pi') \). Then \( \mu' \in \text{Sol}(\Pi') \) and \( \mu'' \in \text{Sol}(\sigma') \).

By definition, \( \mu'' \in \text{Sol}(\sigma') \), and finally \( \mu'' \in \text{Sol}(\Pi' \land \sigma') \).

We have \( \mu'' = \exists \Sigma' \Pi' \sigma' \mu' \). Then \( \mu'' = \exists \Sigma' \Pi' \sigma' \mu \).

Assume now that \( \mu'' = \exists \Sigma' \Pi' \sigma' \mu \).

Finally, since \( \mu'' = \exists \Sigma' \Pi' \sigma' \mu \) and \( \mu'' = \exists \Sigma' \Pi' \sigma' \mu \), then \( \mu \in \text{Sol}(\Pi') \).

It proves \( \text{Sol}(G) \subseteq \text{Sol}(G') \).

Assume that \( \mu \in \text{WTSol}(G) \). There exists \( \mu' = \exists \Sigma' \Pi' \mu \) such that \( \mu' \in \text{WTSol}(P \land C \land M \land H \land \Pi' \land \sigma) \) \( \Pi' \land \sigma \).

Moreover, since \( \Sigma' \) are new variables not occurring in \( \Pi' \land \sigma \), we have \( \mu'' \in \text{Sol}(P \land C \land M \land H \land \Pi' \land \sigma) \), and \( \mu'' \in \text{Sol}(\Pi') \).

Finally, since \( \mu'' = \exists \Sigma' \Pi' \mu \) and \( \mu'' = \exists \Sigma' \Pi' \mu \), then \( \mu \in \text{WTSol}(G) \).

It proves \( \text{WTSol}(G) \subseteq \text{WTSol}(G') \).

**Rule SF, Solving Failure:**

We prove \( \text{WTSol}(G) = \emptyset \) by contradiction, under the assumptions (a) and (b) stated in the theorem.

We assume that there exists \( \mu \in \text{WTSol}(G) \). Then, by definition, there exists \( \mu' = \exists \Sigma' \Pi' \mu \) such that \( \mu' \in \text{WTSol}(P \land C \land M \land H \land \Pi' \land \sigma) \).

We know now that \( \mu' \in \text{Sol}(\Pi') \). Then \( \mu' \in \text{Sol}(\Pi') \) and \( \mu'' \in \text{Sol}(\Pi') \) and \( \mu'' \in \text{Sol}(\Pi') \).

By definition, \( \mu'' \in \text{Sol}(\Pi') \), and finally \( \mu'' \in \text{Sol}(\Pi') \).

We prove \( \text{Sol}(G) = \emptyset \).

Note that the completeness part (item 2) of Theorem 2 also implies that failing goals have no solution; i.e., from a failing transformation step \( G \mapsto R \), we can conclude \( \text{WTSol}(G) = \emptyset \).

The soundness of the goal solving calculus (item a) in Theorem 1) follows easily from the first item of Theorem 2. It is stated and proved in the next corollary, ensuring 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 \( G \).

**Corollary 1 (Soundness).**

Let \( G \) be an initial goal and \( P \) a CFLP(\( \Sigma \))-program such that \( G \vdash \top \Sigma \), where \( S \) is a solved goal. Then, \( \text{Sol}(\Sigma) \subseteq \text{Sol}(G) \).

**Proof.** Using the first item of Theorem 2, an easy induction shows that \( \text{Sol}(G) \subseteq \text{Sol}(G) \) holds for any \( G \) such that \( G \vdash \top \Sigma \). In particular, one gets that \( \text{Sol}(\Sigma) \subseteq \text{Sol}(G) \) for each solved form \( S \) such that \( G \vdash \top \Sigma \). Since \( \text{Sol}(\Sigma) = \text{Sol}(S) \), the corollary is proved.

In order to prove completeness of the goal solving calculus (item b) in Theorem 1) one must assume an arbitrary given \( \mu \in \text{WTSol}(G) \) and ensure the existence of a terminating computation eventually leading to \( \mu \). This is done with the help of a proof technique similar to that used for the CLNLC(D) calculus shown in [14]. Remember that \( \mu \in \text{WTSol}(G) \) implies the existence of a multiset \( M \) of CRW L(\( \Sigma \)) proofs acting as witness. A witness \( M \)
for the fact that \( \mu \in Sol_\chi(G) \) is defined as a multiset containing all the CRW L(C)-proofs mentioned in [14]. We prove Lemma 1 below by using the second item of Theorem 2 and defining a well-founded progress ordering \( \triangleright \) between pairs \((G, M)\) formed by a goal \( G \) and a witness. Next, we prove item (b) of Theorem 1 as an easy corollary of Lemma 1.

**Lemma 1 (Progress Lemma).**

Let \( G \) be an admissible goal for a CFLP(C)-program \( \mathcal{P} \), a well-typed solution \( \mu \in WTSol_\chi(G) \) and a witness \( M \) for it. Assume that neither \( \mathcal{P} \) nor \( G \) have free occurrences of higher-order variables, and that \( G \) is not in solved form. Then:

1. There is some rule applicable to \( G \).
2. For any rule \( RL \) which can be safely applied to \( G \) without involving an incompleteness solver invocation, and without a opaque decomposition step, there exists an admissible goal \( G' \) and a well-typed solution \( \mu' \in WTSol_\chi(G') \) with witness \( M' \) such that:
   - \( G \vdash RL \vdash G' \).
   - \( \mu = \mu' \{ [\nabla'] \} \), where \( \nabla' \) are the new fresh variables of \( G' \) introduced by the \( RL \) step.
   - \((G, M) \triangleright (G', M')\), where \( \triangleright \) is the well-founded progress ordering defined in the proof below.

**Proof.** (1) For the proof of this item we consider the full set of goal transformation rules of our cooperative goal solving calculus, consisting of the rules displayed in Figure 1, plus other rules similar to those presented in [5] that take care of constraint flattening and lazy narrowing. Assume that \( G \equiv \exists \mathcal{U} \).

First, we observe that the failure rules CF and SF cannot be applied to \( G \), because if \( G \vdash RL \vdash G \) then \( WTSol_\chi(G) = \emptyset \), and it is not possible by initial hypothesis of the lemma.

Next, we consider two cases, either \( P \) and \( C \) are empty or not empty. If both, \( P \) and \( C \) are empty, then there exists at least a constraint store, \( M \), \( H \), \( F \) or \( R \) that is not in solved form, then the constraint solver rule \( CS \) is an applicable goal transformation rule.

In other case, if \( C \) is not empty, then two possibilities can be. Either \( C \) contains a no atomic primitive constraint and therefore the rule FC can be applied or else all constraints are flattened, that is, \( C \) only contains atomic primitive constraints \( \pi \) of the form \( p_{\pi} \rightarrow! t \), where \( \pi \) is a bridge, or a Herbrand constraint, or a \( FD \) or \( R \) constraint and this case the rules \( SB \), \( PP \), \( SC \) can be applied.

If \( C \) is empty then \( P \) is not empty, and the rules \( DC \), \( SP \), \( IM \), \( DF \) and \( PC \) are applicable if at least a production in \( P \) has one of the following forms:

- \( h \tau_{m} \rightarrow h \tau_{m} \) where \( h \) is not opaque, in this case the rule \( DC \) can be applied.
- \( s \rightarrow p \), where \( s \equiv X \in \forall r \), and \( t \not\in \forall r \) or \( s \in \forall p_{\tau} \), and \( t \equiv X \in \forall r \), in this case the rule \( SP \) can be applied.
- \( h \tau_{m} \rightarrow X \) with \( h \tau_{m} \not\in \forall p_{\tau} \) is passive, and \( X \) is a demanded variable, in this case the rule \( IM \) can be applied.
- \( f \tau_{m} \rightarrow t \), where \( f \in DF^{n} (k \geq 0) \), \( t \not\in \forall r \) or \( t \) is a demanded variable, in this case the rule \( DF \) can be applied.
- \( p \tau_{n} \rightarrow t \), where \( p \in PF^{n} \), \( t \not\in \forall r \) or \( t \) is a demanded variable, in this case the rule \( PC \) can be applied.

Now, for the rest of productions, we proceed by assuming gradually that no rule, except \( EL \), is applicable to \( G \), and then we will concluded that this remaining rule \( EL \) must be applicable.

We assume that \( DC \), \( SP \), \( IM \), \( DF \) and \( PC \) are not rules applicable. Then, each production in \( P \) is a suspension of one of the following three forms:

\[
h \tau_{m} \rightarrow X, \quad f \tau_{m} \rightarrow X (\text{with } k \geq 0), \quad o \tau_{n} \rightarrow X
\]

Where \( X \) is always a produced but not obviously demanded variable. Whereas \( \chi = pvar_\chi(G) \). As the rule \( CS \) is not applicable, then the constraint stores \( M \), \( H \), \( F \) and \( R \) must be in solved form w.r.t. the set of critical variables. They satisfied the following conditions:

- Since \( pvar_\chi(P) \cap odvar_\chi(P) = \emptyset \), all the produced variables which appear in these constraint stores must be critical, because they are not obviously demanded variables. Let \( \chi \) the set formed by these critical variables.
- We choose the variable \( X \in \chi \) that is minimal w.r.t. the production relation \( \triangleright_{P} \) (such element must exists because of the finite number of variables in the admissible goal \( G \)). By the Discrimination property of the constraint solver, either \( X \) becomes obviously demanded, or else disappear. Since we know that \( X \) is not an obviously demanded variable, by construction of \( \chi \), we deduce that \( X \) cannot appear in any constraint store of \( G \).
- \( X \) does not appear neither the rest of productions of \( P \) (because \( X \) is minimal w.r.t. the production relation \( \triangleright_{P} \)) nor the constraint stores. Then, \( X \) does not appear in the rest of the goal.

Therefore, the rule \( EL \) can be applied on \( G \) over the production in which \( X \) appears as a produced variable.

(2) By applying the second item of Theorem 2, we have \( G \vdash RL \vdash G_k \) for each \( 1 \leq k \leq l \) and \( \mu \in WTSol_\chi(G) \subseteq \bigcup_{k=1}^{l} WTSol_\chi(G_k) \) for any rule \( RL \) which can be safely applied to \( G \) without involving an incompleteness solver invocation. Let \( G' \equiv G_k (1 \leq k \leq l) \) such that \( \mu \in WTSol_\chi(G_k) \). Since the derivation \( G \vdash RL \vdash G' \) does not introduce new free (higher-order) variables, by applying the definition of set of solutions, we deduce that there exits a well-typed solution \( \mu' \in WTSol_\chi(G') \) such that \( \mu = \mu' \{ [\nabla'] \} \), where \( \nabla' \) are the new fresh variables of \( G' \) introduced by the \( RL \) step.

To finish the proof, we define the well-founded progress ordering \( \triangleright \) defined in the proof below. Given the admissible goal \( G \) for the CFLP(C)-program \( \mathcal{P} \) (both without free occurrences of higher-order variables), and the witness \( M \); \( \mu \in WTSol_\chi(G) \), we define the following sizes and measures associated to \( G \) and \( M \):

**O₁** The restricted size of the witness \( M = \{ T_1, \ldots, T_n \} \), corresponding to \( \mu \in WTSol_\chi(G) \), is the multiset of natural numbers \( \{ 1 \mid T_i \} \cup \{ 0 \mid T_i \} \), where \( \{ T_i \} (1 \leq i \leq n) \) denotes the restricted size of a proof tree \( T_i \), \( (1 \leq i \leq n) \) as defined in [15].

**O₂** The number of occurrences in \( G \) of rigid and passive expressions \( h \tau_{n} \) that are not patterns.

**O₃** The total syntactic size of the right hand sides of productions in \( G \).

**O₄** The total restricted size of the constraint stores, where the restricted size of a constraint store is defined as follows: \( 0 \) is the constraint store is in solved form not involving incompleteness solver invocations, and \( 1 \) in other case.

Next, we define the ordering \( \triangleright \) to act over pairs \((G, M)\) as the lexicographic product of \( >_{mul} \times > \times > \times > \), where \( >_{mul} \times > \times > \times > \)
is the multiset order for multisets over the set of natural numbers \( \mathbb{N} \), and \( > \) is the usual ordering over \( \mathbb{N} \). Lexicographic products and the multiset ordering construction are known to preserve the well-founded character of orderings (see e.g. [3]). Therefore, \( \triangleright \) is well-founded.

<table>
<thead>
<tr>
<th>RULES</th>
<th>O₁</th>
<th>O₂</th>
<th>O₁</th>
<th>O₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EL ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DF ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FC ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB + PP, SC ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS ( \geq_{\text{mul}} \geq \geq \geq \geq \geq \geq )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Well-founded progress ordering \( \triangleright \)

Table 1 shows the behavior of the different transformations given in Table 1 and [5] w.r.t. the four components of the well-founded progress ordering \( \triangleright \). It proves that \( (G, M) \triangleright (G', M') \) holds for any goal transformation step \( G \vdash \mu \), and \( G' \), which involves no incomplete solver invocation and no opaque decomposition. This completes the proof of the lemma.

Finally, the completeness of the goal solving calculus (item (b) in Theorem 1) follows easily from Lemma 1. It is stated and proved in the next corollary.

COROLLY 2 (Completeness).

Let \( G \) be an admissible goal for a \( CFLP(C) \)-program \( P \) and \( \mu \in WTSol_P(G) \). Assume that neither \( P \) nor \( G \) have free occurrences of higher-order variables. Then, unless prevented by an incomplete solver invocation or an opaque decomposition step, one can find a computation \( G \vdash \mu \) ending with a goal in solved form \( S \), and there is some \( \mu' \in WTSol_P(S) \) such that \( \mu = \mu'[\text{var}(G)] \).

PROOF 4. In case that \( G \) is in solved form, the theorem can be trivially proved by taking \( S = G \) and \( \mu = \mu' \). Otherwise, we can apply Lemma 1 to obtain (unless prevented by an incomplete solver invocation and an unsafe rule application) a step \( G \vdash \mu_1 \) and \( \mu_1 \in WTSol_P(G_1) \) with witness \( M_1 \), such that \( \mu = \mu_1[V_1^0] \) (where \( V_1^0 \) are the new fresh variables introduced by the step) and \( (G, M) \triangleright (G_1, M_1) \). The goal \( G_1 \) can have no free occurrences of higher-order variables, because \( G_1 \) and all rules trivially preserve this property.

Hence, the hypothesis of the theorem apply to the solution \( \mu_1 \in WTSol_P(G_1) \) but with \( (G, M) \triangleright (G_1, M_1) \). Reasoning by induction on the well-founded ordering \( \triangleright \) (see e.g. [3] for an explanation of this proof technique) we can assume that (unless prevented by some incomplete solver invocation and unsafe rule application) there is a computation \( G_1 \vdash \mu_2 \) ending with a goal in solved form \( S \), and some \( \mu' \in WTSol_P(S) \) such that \( \mu_1 = \mu'[V_1^0] \), where \( V_1^0 \) collect all the new fresh variables introduced by the computation. Then, \( V_1^0 = \mu'V_1^0 \) and \( V_1^0 \) collect all the new fresh variables introduced by the computation \( G \vdash \mu \) \( G_1 \vdash \mu_2 \) \( S \), and \( \mu = \mu'[V_1^0] \) trivially follows from \( \mu = \mu_1[V_1^0] \) and \( \mu_1 = \mu'[V_1^0] \). This finishes the proof.

4. Implementation/Performance

We have implemented our cooperative computation model for \( CFLP(C) \) in the \( TOY \) system [1], which is implemented in SICStus Prolog 3.11 [18]. In this section we provide some details about implementation issues, and also have worked out a set of benchmarks with two aims: first, to compare the performance of our system with the closest related system we are aware of, namely META-S [6]; and, second, to study how the performance of \( TOY \) is affected by the cooperation mechanisms, with special regard to the enabled solvers and their cooperation.

4.1 \( TOY \) vs META-S

As already indicated, our implementation has been developed in the \( TOY \) system [1], which is implemented in SICStus Prolog 3.11. \( TOY \) already supported non-cooperative \( CFLP \) programming over \( H, FD, R \), using the \( FD \) and \( R \) solvers provided by SICStus along with Prolog code for the \( H \) solver. This former system has been extended to support cooperative \( CFLP \) computation, including a store for \( M \)-constraints and implementing mechanisms for computing bridges and projections. Since the numerical equivalence associated to bridges may be not exact, the system uses a user-defined tolerance parameter.

Our cooperative proposal can be compared with the approach used to construct META-S, a flexible metasolver that implements the framework for solver cooperation described in [10]. As mentioned in Section 1, we have borrowed the idea of constraint projection from [10], introducing bridge constraints with the aim of making projections more flexible and compatible with type discipline. \( TOY \) system and META-S have some similarities as implemented systems. For instance, META-S also provides communication among several solvers. In particular, three constraint solvers are integrated in META-S for their cooperation:

- a \( FD \) solver (for floats, strings, and rationals) that was implemented in Java using as reference a library of routines for solving binary constraint satisfaction problems provided by Peter van Beek [19];
- a solver for linear arithmetic, i.e., the constraint solver LINEAR described in [13]. This solver is based on the Simplex algorithm and was implemented in the language C. This solver handles linear equations, inequalities, and disequations over rational numbers;
- and an interval arithmetic solver, implemented in Java on the basis of the solver for interval arithmetic from Timothy J. Hickey from Brandeis University [9].

However, there are also significant differences between both systems, concerning the underlying syntax, the implementation techniques and their facilities. In particular, META-S is implemented in Common Lisp whereas \( TOY \) is implemented in Prolog. Also, the solvers attached to \( TOY \) are used as provided by SICStus Prolog 3.1 and they were not internally adjusted to work in a cooperation system, whereas the solvers used in META-S were implemented with regard to their integration into the implementation of META-S as a system with cooperating components. Another significant difference is that \( TOY \) provides constraint optimization mechanisms, whereas META-S does not.

4.2 Benchmarks and Results

For the comparison, we have executed a set of benchmarks, using the last version of META-S (kindly provided by its implementors
on our request), compiled using CMUCL 18d. Our set of benchmarks is reasonably wide. For the sake of fairness, whenever it was possible, we used exactly the same formulation of the problems as well as the same constraints.

All the programs used in the comparison are available at http://www.lcc.uma.es/~afdez/cflpfdr/. The benchmarks are:

Sendmore (smm): A cryptoarithmethic problem with 8 variables ranging over [0,9], one linear equation, two disequations and one all different constraint.

Non-linear crypto-arithmetic (nl-csp): A problem with 9 variables and non-linear equations.

Wrong-Wright (wwr): A cryptoarithmethic problem with 8 variables ranging over [1,9], one linear equation, and one all different constraint.

3 × 3 Magic Square (mag.sq): A problem that involves 9 FD variables and 7 linear equations.

Equation 10 (eq10): A system of 10 linear equations with 7 variables ranging over [0,10].

Equation 20 (eq20): As eq10 but for 20 linear equations.

Intersection (bothIn): Computing the intersection of a triangle and a discrete grid (as in the goal at the end of Subsection 3.1) with $n = 800$, $d = 400$.

An electrical circuit (circuit): An electric circuit with some connected resistors (modelled with real variables) and a set of capacitors (modelled with FD variables). The goal consists of finding the capacitor that has to be used, so that its voltage reaches the 99% of the final voltage between a time range.

The formulation of benchmarks nl-csp, mag.sq, circuit, and smm has been taken from META-S’ distribution. All the benchmarks were executed on the same Linux machine (under professional Suse Linux 9.3) with an Intel Pentium M processor running at 1.70GHz and with a RAM memory of 1 GB. For the sake of brevity, we only provide the results for first solution search, which are listed in Figure 2, showing the average running time of ten runs (measured in milliseconds). The upper part of the table corresponds to META-S. For the performance analysis, at least the FD solver is needed because all the benchmarks need FD variables. In addition, an arithmetic solver is also needed in certain benchmarks (i.e., bothIn and circuit). We have selected the linear arithmetic solver since the interval arithmetic solver yielded poorer results in all cases. In addition, we have considered the best problem formulation (in terms of the target solver for each constraint) that yielded the best running time.

For this system, the first column of the table displays four resolution strategies that META-S provides: eager, in which all constraint information is propagated as early as possible in the solving process; heuristic, in which additionally the alternative (i.e., the value of a variable) that is most likely to lead to an inconsistency is processed first; and eager-ordered and heuristic-ordered, which respectively correspond to eager and heuristic with the addition of an order for the projection of variables. Labeling is implicit in all these strategies.

The rest of the table shows the results for TOY and we have considered three configurations: a) TOY(FD), i.e., TOY with the FD solver enabled. b) TOY(FD + R), i.e., TOY with both solvers FD and R enabled and projection disabled, and c) TOY(FD + R)-proj, i.e., as TOY(FD + R) but with projection enabled. Note that the H solver in TOY is build-in, and therefore it is always enabled.

In addition, whenever possible, we have considered two alternative TOY formulations of the benchmarks: one with arithmetic constraints expressed in the finite domain (labelled with FD in the table), and the other one in the real domain (labelled with $\sim R$). (Incompatible combinations are not shown, as e.g. TOY(FD) with both solvers FD and $\sim R$ enabled and projection disabled, and TOY(FD + R)-proj, i.e., as TOY(FD + R) but with projection enabled. Note that the H solver in TOY is build-in, and therefore it is always enabled.

In TOY there is currently no option for choosing different resolution strategies, but one can use built-in labeling strategies and define new ones. We have considered built-ins: naïve, in which variables are labelled in a prefix order (i.e., the leftmost variable is selected); and first fail (ff), in which the variable with the smallest domain is chosen first for enumerating. In certain form, naïve and ff labeling strategies in TOY are similar, respectively, to eager and heuristic strategies in META-S. To keep a higher similarity, we also took care that the variable orders were identical for the different resolution/labelling strategies in both systems. For both systems (META-S and TOY), we highlight in bold face the best result obtained for solving a given benchmark for its different configurations. Finally, N/A indicates that a benchmark could not be formulated in TOY using only finite domain constraints.

<table>
<thead>
<tr>
<th>Strategy/Version</th>
<th>smm</th>
<th>nl-csp</th>
<th>wwr</th>
<th>mag.sq</th>
<th>eq10</th>
<th>eq20</th>
<th>bothIn</th>
<th>circuit</th>
</tr>
</thead>
<tbody>
<tr>
<td>META-S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eager</td>
<td>950</td>
<td>348400</td>
<td>10930</td>
<td>1160</td>
<td>60</td>
<td>60</td>
<td>9090</td>
<td>70</td>
</tr>
<tr>
<td>eager.ordered</td>
<td>620</td>
<td>1230</td>
<td>650</td>
<td>1220</td>
<td>60</td>
<td>60</td>
<td>9120</td>
<td>70</td>
</tr>
<tr>
<td>heuristic</td>
<td>590</td>
<td>302314</td>
<td>620</td>
<td>520</td>
<td>70</td>
<td>70</td>
<td>9100</td>
<td>70</td>
</tr>
<tr>
<td>heuristic.ordered</td>
<td>580</td>
<td>970</td>
<td>620</td>
<td>540</td>
<td>70</td>
<td>70</td>
<td>9120</td>
<td>70</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>16</td>
<td>15</td>
<td>18</td>
<td>92</td>
<td>74</td>
<td>138</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>15</td>
<td>20</td>
<td>19</td>
<td>91</td>
<td>90</td>
<td>134</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>89</td>
<td>75</td>
<td>139</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>22528</td>
<td>411</td>
<td>411</td>
<td>166</td>
<td>266</td>
<td>402</td>
<td>110</td>
<td>14</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>89</td>
<td>81</td>
<td>131</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>22527</td>
<td>383</td>
<td>420</td>
<td>168</td>
<td>271</td>
<td>408</td>
<td>110</td>
<td>13</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>47</td>
<td>39</td>
<td>58</td>
<td>87</td>
<td>284</td>
<td>431</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>41</td>
<td>44</td>
<td>54</td>
<td>158</td>
<td>290</td>
<td>433</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>49</td>
<td>86</td>
<td>52</td>
<td>91</td>
<td>261</td>
<td>421</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>F</td>
<td>40</td>
<td>87</td>
<td>58</td>
<td>163</td>
<td>269</td>
<td>397</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 2. Performance Results: META-S vs. TOY
The best results in Figure 2 are summarized in Figure 3, which also shows the speed-up of TOY wrt. META-S for each of the benchmarks, as well as the average speed-up.

4.3 Analysis of the Results

With respect to the TOY results, note that for those benchmarks more naturally coded using FD constraints (as smm, wwr and mag.sq), the best results are obtained by activating only the FD solver (see rows for TOY(FD)). In these cases, however, no appreciable overhead is noticed when the R solver is also activated (compare for instance rows for TOY(FD) and TOY(FD + R) with those for TOY(FD) with naïve FD and fl FD configurations). On the other hand, enabling the projection mechanism in TOY leads in certain cases to significant improvements, as for the benchmark bothIn (see rows for TOY(FD + R) and TOY(FD + R)-proj with naïve FD ∼ R and fl FD ∼ R). This is because projections from R to FD drastically reduce the domains of FD variables, as discussed in Section 1. Regarding the comparison to META-S, we have to say that META-S seems to behave particularly well in the solving of linear equations, specially when the problem requires no global constraints (such as an all different constraint used in benchmarks eq10 and eq20). The reason maybe that the linear arithmetic solver of META-S is more optimized than its FD solver. However, in general TOY shows an improvement of about one order of magnitude (considering the listed set of benchmarks) with respect to the META-S system. Although these results suggest that our proposal is not only promising but also interesting in its current state, we are aware that our system should be extended to support more powerful projections and strategies. Also, a more extensive benchmark analysis will be necessary in the future, in order to obtain determinant conclusions.

5. Conclusions and Future Work

We have investigated foundational and practical issues concerning a computation framework for the cooperation of domains in CFLP, using projections guided by bridge constraints. A selection of related work has been cited in Section 1 and some more detailed comparisons have been given in [5]. Our results include both the formal computational framework described in Sections 2 and 3 and the implementation reported in Section 4, showing a good performance with respect to the META-S system. The soundness and completeness theorem in Section 3 and the benchmarking results in Section 4 are presented here for the first time.

As interesting lines of future work we foresee: expanding the implementation to support other coordination domains, considering Boolean constraints, set constraints, or numeric constraints over different numeric types; implementing more powerful projections acting over the constraints inside the constraint stores; and designing useful strategies for controlling the application of goal transformation rules.

Acknowledgments

This research has been partially supported by the Spanish National Project MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS–CAM(S-0505/TIC/0407).

<table>
<thead>
<tr>
<th>System</th>
<th>smm</th>
<th>nl-csp</th>
<th>wwr</th>
<th>mag.sq</th>
<th>eq10</th>
<th>eq20</th>
<th>bothIn</th>
<th>circuit</th>
<th>Avg.Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>META-S</td>
<td>380</td>
<td>970</td>
<td>620</td>
<td>520</td>
<td>60</td>
<td>60</td>
<td>9090</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>TOY</td>
<td>14</td>
<td>15</td>
<td>18</td>
<td>87</td>
<td>74</td>
<td>131</td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>META-S/TOY</td>
<td>41.42</td>
<td>66.66</td>
<td>34.44</td>
<td>5.97</td>
<td>0.81</td>
<td>0.45</td>
<td>9.09</td>
<td>5.38</td>
<td>51.09</td>
</tr>
</tbody>
</table>

Figure 3. Best Results and Speed-Ups

Last author was partially supported by Spanish MCyT projects under contracts TIN2005-08818-C04-01 and TIN-2007-67134. The authors are also grateful to the anonymous reviewers for their constructive comments and suggestions.

References

