Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 269 (2011) 55–69
www.elsevier.com/locate/entcs

A Theoretical Framework for the Higher-Order Cooperation of Numeric Constraint Domains

Rafael del Vado Vírseda

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Spain
rdelvado@sip.ucm.es

Abstract

This paper presents a theoretical framework for the integration of the cooperative constraint solving of numeric constraint domains into higher-order functional and logic programming on λ-abstractions, using an instance of a generic Constraint Functional Logic Programming (*CFLP*) scheme over a so-called higher-order coordination domain. We provide this framework as a powerful computational model for the higher-order cooperation of algebraic constraint domains over real numbers and integers, which has been useful in practical applications involving the hybrid combination of its components, so that more declarative and efficient solutions can be promoted. Our proposal of computational model has been proved sound and complete with respect to the declarative semantics provided by the *CFLP* scheme, and enriched with new mechanisms for modeling the intended cooperation among the numeric domains and a novel higher-order constraint domain equipped with a sound and complete constraint solver for solving higher-order equations. We argue the applicability of our approach describing a prototype implementation on top of the constraint functional logic system $\mathcal{TOY}$.

Keywords: higher-order cooperation, constraint domains, functional and logic programming

1 Introduction

The effort to identify suitable theoretical frameworks for higher-order functional logic programming has grown in recent years [2,9,13,14,16]. The high number of approaches in this area and their different scopes and objectives indicate the high potential of such a paradigm in modeling complex real-world problems [13]. Functional logic programming is the result of integrating two of the most successful declarative programming styles, namely functional and logic programming, in a way that captures the main advantages of both [3]. Whereas higher-order programming is standard in functional programming, logic programming is in large part still tied to the first-order world. Only a few higher-order logic programming languages, most notably λ-*Prolog* [10], use higher-order logic for logic programming and have shown its practical utility, although the definition of evaluable functions is

doi:10.1016/j.entcs.2011.03.005

not supported. Moreover, higher-order constructs such as function variables and λ-abstractions of the form $\lambda x.\, e$ (the syntax stands for an anonymous function which, when given any actual parameter in place of the formal parameter x, will return the value resulting from the evaluation of the body expression e) are widely used in functional programming and higher-order logic programming languages, where λ-terms are used as data structures to obtain more of the expressivity of higher-order functional programming.

Within this research area, we have proposed in [15,16] a complete theoretical framework for higher-order functional logic programming as an extension to the setting of the simply typed lambda calculus of a first-order rewriting logic, where programs are presented by *Conditional Pattern Rewrite Systems* (*CPRS* for short) on lambda abstractions. For a first impression of our higher-order programming framework, the following *CPRS* illustrates the syntax of patterns on lambda abstractions to define a classical higher-order function *map* for the application of a given function to a list of elements.

$$\begin{aligned} map\,(\lambda u.\, F(u), [\,]) &= [\,] \\ map\,(\lambda u.\, F(u), [\,X \mid Xs\,]) &= [\,F(X) \mid map\,(\lambda u.\, F(u), Xs)\,] \end{aligned}$$

The first aim of this paper is to present a theoretical framework for the integration of higher-order functional logic programming with *constraint solving*, extending our programming language with the capacity of solving constraints over a given algebraic constraint domain. The term *constraint* is intuitively defined as a relationship required to hold among certain entities as variables and values (e.g., $X + Y \leq 0$). We can take for instance the set of integers or the set of real numbers with addition, multiplication, equality, and perhaps other functions and predicates. Among the formalisms for the integration of constraints in functional logic programming we use in this work the *Constraint Functional Logic Programming* scheme $CFLP(\mathcal{D})$ [7] which supports a powerful combination of functional and constraint logic programming over $\mathcal{D}$ and can be instantiated by any constraint domain $\mathcal{D}$ given as parameter which provides specific data values, constraints based on specific primitive operations, and a dedicated constraint solver. There are different *instances* of the scheme for various choices of $\mathcal{D}$, providing a declarative framework for any chosen domain $\mathcal{D}$. Useful constraint domains include the *Herbrand domain* $\mathcal{H}$ which supplies equality and disequality constraints over symbolic terms, the algebraic domain $\mathcal{R}$ which supplies arithmetic constraints over real numbers, and the algebraic domain $\mathcal{FD}$ which supplies arithmetic and finite domain constraints over integers. As a concrete example of a *CPRS* integrating higher-order functional logic programming with algebraic constraints in $\mathcal{R}$, we can consider the following variant of a classical higher-order function *diff* to compute the differential of a function f at some numeric value X under some arithmetic constraints over real numbers in the conditional part of program rules.

$$diff :: (real \to real) \to real \to real$$

$$
\begin{array}{lcll}
\mathit{diff}\ (\lambda u.\, u, X) & = & 1 & \\
\mathit{diff}\ (\lambda u.\, \mathit{sin}\ (F(u)), X) & = & \mathit{cos}\ (F(X)) * \mathit{diff}\ (\lambda u.\, F(u), X) & \Leftarrow \quad \pi/4 \leq F(X) \leq \pi/2 \\
\mathit{diff}\ (\lambda u.\, \mathit{ln}\ (F(u)), X) & = & \mathit{diff}\ (\lambda u.\, F(u), X)/F(X) & \Leftarrow \quad F(X) \neq 0
\end{array}
$$

In contrast to first-order programming, we can easily formalize functions to be differentiated, or to compute the inverse operation of the differentiation (integration) by means of *narrowing* [15] as a suitable operational semantics, a transformation rule which combines the basic execution mechanism of functional and logic languages, namely *rewriting* with *unification*. For instance, we can compute by narrowing the substitution $\{F \mapsto \lambda u.\, \mathit{sin}\ (u)\}$ as a solution of the goal $\lambda x.\, \mathit{diff}\ (\lambda u.\mathit{ln}\ (F(u)), x) == \lambda x.\, \mathit{cos}\ (x)/\mathit{sin}\ (x)$ because the constraint $\lambda x.\, (\pi/4 \leq x \leq \pi/2 \rightarrow \mathit{sin}\ (x) \neq 0)$ is evaluated to *true* by an $\mathcal{R}$-constraint solver.

Practical applications in higher-order functional logic programming, however, often involve more than one "pure" domain (i.e., $\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$, etc.), and sometimes problem solutions have to be artificially adapted to fit a particular choice of domain and solver. 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 constraint programming community. An important idea emerging from the research in this area is that of "hybrid" constraint domain (e.g., $\mathcal{H} \oplus \mathcal{R} \oplus \mathcal{FD}$ [1]), 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.

2 Higher-Order Algebraic Constraint Cooperation

The second contribution of this work is to present a formal framework for the cooperation of the algebraic constraints domains $\mathcal{FD}$ and $\mathcal{R}$ in an improved version of the $\mathit{CFLP}(\mathcal{D})$ scheme [7], now useful for higher-order functional and logic programming on lambda abstractions. As a result, we provide a powerful theoretical framework for higher-order constraint functional logic programming with lambda abstractions and decidable higher-order unification in a new higher-order constraint domain $\diagdown$, which leads to greater expressivity. As a motivation for the rest of the paper, we present in this section a couple of examples of *CPRS*-programs involving the cooperation of the algebraic constraint domains $\mathcal{FD}$ and $\mathcal{R}$ to illustrate the different cooperation mechanisms that are supported by our theoretical framework, as well as the benefits resulting from the cooperation in the higher-order functional logic programming setting.

As a first simple example, we consider the following *CPRS* (adapted from [1] to the higher-order setting on λ-abstractions) to solve the problem of searching for a two-dimensional point lying in the intersection of a discrete grid and a continuous region.

$$\mathit{bothIn} :: (\mathit{real} \rightarrow \mathit{real} \rightarrow \mathit{real}) \rightarrow \mathit{int} \rightarrow \mathit{int} \rightarrow \mathit{bool}$$

$$\begin{array}{lcll} \mathit{bothIn}\,(\lambda u, v.\, F(u,v), X, Y) & = & \mathit{true} & \Leftarrow \quad X \rightleftharpoons RX,\, Y \rightleftharpoons RY, \\ & & & \quad F(RX, RY) \leq 0, \\ & & & \quad \mathit{domain}\ [X,Y]\ 0\ N,\ \mathit{labeling}\ [\,]\ [X,Y] \end{array}$$

The higher-order function *bothIn* is intended to check if a given discrete point (X, Y) belongs to the intersection of the continuous region given by the $\mathcal{R}$-constraint $F(RX, RY) \leq 0$ and the discrete grid given by the $\mathcal{FD}$-constraints *domain* $[X, Y]$ 0 N, *labeling* $[\,]$ $[X, Y]$, ensuring that the variables X and Y are bound to integer values in the interval $[0..N]$. In order to model the intended cooperation and communication between the constraint domains $\mathcal{FD}$ and $\mathcal{R}$ we use a special kind of hybrid constraints $\rightleftharpoons$ called *bridges*, as a key tool for communicating constraints between different algebraic constraint domains. More precisely, the two communicating constraints $X \rightleftharpoons RX$ and $Y \rightleftharpoons RY$ ensure that the discrete point (X, Y) and the continuous point (RX, RY) are equivalent. Different goals can be posed and solved using the small program just described. For instance, the goal $\mathit{bothIn}\,(\lambda u, v.\, v - 4 * u + u^2, X, Y) == \mathit{true}$ with $N = 4$ asks for points in the intersection of the square grid with the inner side of the parabola $Y = 4 * X - X^2$. We can compute by narrowing 15 solutions (see **Fig. 1**): $\{X \mapsto 2, Y \mapsto 3\}$, $\{X \mapsto 1, Y \mapsto 2\}$, $\{X \mapsto 2, Y \mapsto 2\}$, $\{X \mapsto 3, Y \mapsto 2\}$, etc. In this process, cooperation between the $\mathcal{R}$-constraint solver and the $\mathcal{FD}$-solver is crucial for the efficiency of the computation. Initially, we reduce the problem of solving the goal to the problem of solving the hybrid constraint system $\{\, X \rightleftharpoons RX,\ Y \rightleftharpoons RY,\ RY - 4 * RX + RX^2 \leq 0,\ \mathit{domain}\ [X, Y]\ 0\ 4,\ \mathit{labeling}\ [\,]\ [X, Y]\,\}$. When the communication constraints are disabled, the last $\mathcal{FD}$-constraints force the enumeration of all possible values for X and Y within their domains, eventually finding all the solutions after $\mathcal{O}(N^2)$ steps. When the communication constraints are enabled, we can use both constraints to project the $\mathcal{R}$-constraint $RY - 4 * RX + RX^2 \leq 0$ into equivalent *integer* $\mathcal{FD}$-constraints $(X = 0) \wedge (Y = 0)$, $(X = 1) \wedge (0 \leq Y \leq 3)$, $(X = 2) \wedge (0 \leq Y \leq 4)$, $(X = 3) \wedge (0 \leq Y \leq 3)$, $(X = 4) \wedge (Y = 0)$. Now, using this new information the $\mathcal{FD}$-solver can prune the domains of X and Y, and solving the labeling constraint leads to the solutions with minor effort. The expected speedup in execution time corresponds to the improvement from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ or $\mathcal{O}(1)$ steps according to the lambda abstraction encoded in the goal and the possibilities offered by the constraint solver.

We present now a second example, intended to illustrate new possibilities and mechanisms of our higher-order cooperative constraint model. In engineering, a common problem is the approximation of a complicated continuous function by a simple discrete function (e.g., the approximation of *GPS* satellite coordinates). Suppose we know a real function (given by a lambda abstraction $\lambda u.\, F(u)$) but it is too complex to evaluate efficiently. Then we could pick a few approximated (integer) data points from the complicated function, and try to interpolate those data points to construct a simpler function, for example, a polynomial $\lambda u.\, P(u)$. Of course, when using this polynomial function to calculate new (real) data points we usually do not receive the same result as when using the original function, but depending on the problem domain and the interpolation method used the gain in

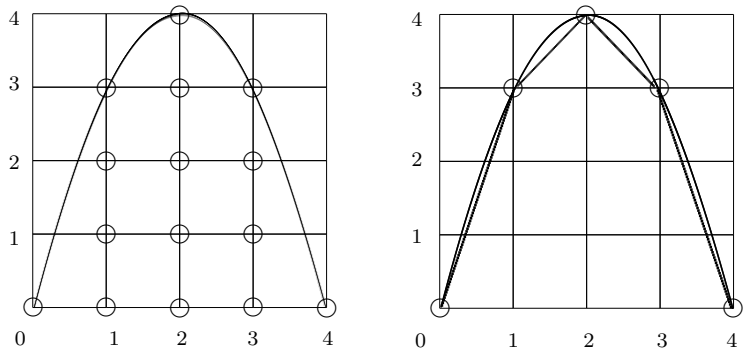

Fig. 1. Cooperation and interpolation of a parabolic region in a discrete square grid.

simplicity might offset the error.

$$
\begin{array}{lcll}
\mathit{disc} :: (\mathit{real} \to \mathit{real}) \to (\mathit{int} \to \mathit{int}) & & & \\
\mathit{disc}\,(\lambda u.\, F(u)) & = & \lambda u.\, P(u) & \Leftarrow \quad \mathit{domain}\ [X]\ 0\ N,\ \mathit{labeling}\ [\mathit{ff}]\ [X], \\
 & & & \quad X \rightleftharpoons RX,\ Y \rightleftharpoons RY, \\
 & & & \quad |\, F(RX) - RY \,| < 1, \\
 & & & \quad \mathit{collection}\ [X,Y]\ C,\ \mathit{interpolation}\ [\mathit{lg}]\ C\ P
\end{array}
$$

Therefore, the aim of this example is to approximate a continuous function represented by a lambda abstraction $\lambda u.\, F(u)$ over real numbers by a discrete polynomial function $\lambda u.\, P(u)$ over integer numbers. In this case, we use the $\mathcal{FD}$-constraints *domain* $[X]$ 0 N, *labeling* $[\mathit{ff}]$ $[X]$ to generate each value of the discrete interval $[0..N]$, according to a *first-fail* (or *ff*) labeling option [7]. The first bridge constraint $X \rightleftharpoons RX$ maps each integer value of X into an equivalent real value in RX. By applying the higher-order functional variable F to RX we obtain the $\mathcal{R}$-constraint $|\, F(RX) - RY \,| < 1$. From this constraint, the $\mathcal{R}$-solver computes (infinite) real values for RY. However, because of the second bridge constraint $Y \rightleftharpoons RY$, each real value assigned to RY by the constraint solving process causes the variable Y to be bound only to an equivalent integer value. By means of the primitive constraint *collection* $[X,Y]$ C we can collect all the pairs (X,Y) generated by the labeling-solving process into a set C. Finally, *interpolation* $[\mathit{lg}]$ C P finds a polynomial which goes exactly through the points collected in C by means of the *Lagrange Interpolation* (*lg*) method. For instance, we can consider the following goal $\mathit{disc}\,(\lambda u.\, 4{*}u{-}u^2)$ $==$ $\lambda u.\, P(u)$ involving the continuous function F as $\lambda u.\, 4 * u - u^2$ with $N = 4$. We obtain the set of integer pairs (x_i, y_i) in $C = \{(0,0),(1,3),(2,4),(3,3),(4,0)\}$ (see again **Fig. 1**). For this particular case, it is easy to check that this computed answer is simply $\{P \mapsto \lambda u.\, 4 * u - u^2\}$.

As we have commented before, the generic scheme $\mathit{CFLP}(\mathcal{D})$ presented in [7] serves in this work as a logical and semantic framework for lazy Constraint Functional Logic Programming over a parametrically given constraint domain $\mathcal{D}$. In order to model the coordination of algebraic constraint domains in our higher-order functional logic programming framework [15,16], we propose the construction of a higher-order *coordination domain* $\mathcal{C}$, as a special kind of hybrid domain tailored to the cooperation of the algebraic domains $\mathcal{R}$ and $\mathcal{FD}$ with a new higher-order constraint domain $\diagdown$ which supplies lambda abstractions as data values and equalities

over lambda terms as constraints. Following the methodology of [1], we obtain a suitable theoretical framework for the cooperation of algebraic constraint domains with their respective solvers in higher-order functional and logic programming using instances $CFLP(\mathcal{C})$. Moreover, thanks to this fact, we can describe a prototype implementation following the techniques summarized in our previous work [1] in the $\mathcal{TOY}$ system [6], which is in turn implemented on top of *SICStus Prolog*. The former system is extended, including special stores for bridges and lambda abstractions, and implementing mechanism for computing bridges, projections, and interpolations according to the new $CFLP(\mathcal{C})$ computation model.

To finish this introduction and motivation to the work, we summarize the organization of the rest of the paper. In Section 3 we give a mathematical formalization of the higher-order constraint domain $\diagdown$ and a solver to solve higher-order strict equations tailored to the needs of the $CFLP(\mathcal{D})$ generic scheme. Followed by a brief presentation of the algebraic constraint domains $\mathcal{R}$ and $\mathcal{FD}$, in Section 4 we discuss bridge constraints and the construction of the coordination domain $\mathcal{C}$ tailored to the cooperation of the algebraic constraint domains $\mathcal{R}$ and $\mathcal{FD}$ with $\diagdown$. In Section 5 we present our proposal of a sound and complete computational model for cooperative higher-order declarative programming in $CFLP(\mathcal{C})$, and we sketch the implementation on top of the $\mathcal{TOY}$ system. Finally, Section 6 summarizes some conclusions and presents a brief outline of related and planned future work.

3 A Higher-Order Constraint Domain

Taking the generic scheme $CFLP(\mathcal{D})$ as a formal basis for foundational and practical issues concerning the cooperation of algebraic constraint domains, in this section we focus on the formalization of a *higher-order constraint domain* $\diagdown$ which supplies λ-abstractions and equality constraints over λ-terms in the instance $CFLP(\diagdown)$. First, we introduce the basic preliminary notions of our higher-order theoretical framework to formalize the constraint domain $\diagdown$ along with a suitable $\diagdown$-*constraint solver* based on an approach similar to the Huèt's procedure of higher-order pre-unification [9,13].

3.1 Preliminary notions

We assume the reader is familiar with the notions and notations pertaining to λ-*calculus* (see, e.g., [5,13] for more examples and motivations). The set of types for simply typed λ-terms is generated by a set $\mathcal{B}$ of *base types* (as e.g., *bool*, *real*, *int*) and the function type constructor "$\rightarrow$". Simply typed λ-*terms* are generated in the usual way from a signature $\mathcal{F}$ of *function symbols* and a countably infinite set $\mathcal{V}$ of *variables* by successive operations of abstraction and application. We also consider the enhanced signature $\mathcal{F}_\perp = \mathcal{F} \cup \mathtt{Bot}$, where $\mathtt{Bot} = \{\perp_b \mid b \in \mathcal{B}\}$ is a set of distinguished $\mathcal{B}$-typed constants. The constant $\perp_b$ is intended to denote an *undefined value* of type b. We employ $\perp$ as a generic notation for a constant from $\mathtt{Bot}$. A sequence of syntactic objects $o_1, \ldots, o_n$, where $n \geq 0$, is abbreviated by $\overline{o_n}$. For instance, the simply typed λ-term $\lambda x_1, \ldots, \lambda x_k.\, (\cdots (a\ t_1) \cdots\ t_n)$ is abbreviated by $\lambda \overline{x_k}.\, a(\overline{t_n})$. Substitutions $\gamma \in \mathit{Subst}(\mathcal{F}_\perp, \mathcal{V})$ are finite type-preserving mappings

from variables to λ-terms, denoted by $\{\overline{X_n \mapsto t_n}\}$, and extended homomorphically to λ-terms. By convention, we write $\{\}$ for the *identity substitution*, $t\gamma$ instead of $\gamma(t)$, and $\gamma\gamma'$ for the function composition $\gamma' \circ \gamma$. The long $\beta\eta$-normal form of a λ-term t, denoted by $t\updownarrow^{\eta}_{\beta}$, is the η-expanded form of the β-normal form of t. It is well-known that $s =_{\alpha\beta\eta} t$ if $s\updownarrow^{\eta}_{\beta} =_{\alpha} t\updownarrow^{\eta}_{\beta}$ [5]. Since $\beta\eta$-normal forms are always defined, we will in general assume that λ-terms are in long $\beta\eta$-normal form and are identified modulo α-conversion. For brevity, we may write variables and constants from $\mathcal{F}$ in η-normal form, e.g., X instead of $\lambda\overline{x_k}.\, X(\overline{x_k})$. We assume that the transformation into long $\beta\eta$-normal form is an implicit operation, e.g., when applying a substitution to a λ-term. With these conventions, every λ-term t has an unique long $\beta\eta$-normal form $\lambda\overline{x_k}.\, a(\overline{t_n})$, where $a \in \mathcal{F}_{\perp} \cup \mathcal{V}$ and $a()$ coincides with a. The symbol a is called the *root* of t and is denoted by $hd(t)$. We distinguish between the set $\mathcal{T}(\mathcal{F}_{\perp}, \mathcal{V})$ of *partial* λ-terms and the set $\mathcal{T}(\mathcal{F}, \mathcal{V})$ of *total* λ-terms. The set $\mathcal{T}(\mathcal{F}_{\perp}, \mathcal{V})$ is a poset with respect to the *approximation ordering* $\sqsubseteq$, defined as the least partial ordering such that:

$$\lambda\overline{x_k}.\, \perp \sqsubseteq \lambda\overline{x_k}.\, t \qquad t \sqsubseteq t \qquad \frac{s_1 \sqsubseteq t_1 \ \cdots \ s_n \sqsubseteq t_n}{\lambda\overline{x_k}.\, a(\overline{s_n}) \sqsubseteq \lambda\overline{x_k}.\, a(\overline{t_n})}$$

A *pattern* [9] is a λ-term t for which all subterms $t|_p = X(\overline{t_n})$, with $X \in \mathcal{FV}(t)$ a *free variable* of t and $p \in MPos(t)$ a *maximal position* in t, satisfy the condition that $t_1{\downarrow_\eta}, \ldots, t_n{\downarrow_\eta}$ is a sequence of distinct elements of the set $\mathcal{BV}(t,p)$ of *bound variables* abstracted on the path to position p in t. Moreover, if all such subterms of t satisfy the additional condition $\mathcal{BV}(t,p) \setminus \{t_1{\downarrow_\eta}, \ldots, t_n{\downarrow_\eta}\} = \emptyset$, then the pattern t is *fully extended*. It is well known that unification of patterns is decidable and unitary [9]. Therefore, for every $t \in \mathcal{T}(\mathcal{F}_{\perp}, \mathcal{V})$ and pattern π, there exists at most one matcher $matcher(t, \pi)$ between t and π.

3.2 The higher-order constraint domain $\diagup$

Intuitively, a *constraint domain* $\mathcal{D}$ provides data values and constraints oriented to some particular application domain. In our higher-order setting, we need to formalize a special *higher-order constraint domain* $\diagup$ to support computations with symbolic equality over λ-terms of any type. Formally, it is defined as follows:

Definition 3.1 [$\diagup$-constraint domain] The *higher-order domain* $\diagup$ is a structure $\langle D_{\diagup}, ==^{\diagup}\rangle$ such that the carrier set $D_{\diagup}$ coincides with the set of ground patterns (i.e., patterns without free variables) over any type, and the function symbol $==$ is interpreted as *strict equality* over $D_{\diagup}$, so that for all $t_1, t_2, t \in D_{\diagup}$, one has $==^{\diagup} \subseteq D^2_{\diagup} \times D_{\diagup}$, where $t_1 ==^{\diagup} t_2 \to t$ (i.e., $(t_1, t_2, t) \in ==^{\diagup}$) iff some of the following three cases hold:

(1) t_1 and t_2 are one and the same total λ-term in $D_{\diagup}$, and *true* $\sqsupseteq t$.

(2) t_1 and t_2 have no common upper bound in $D_{\diagup}$ with respect to the approximation ordering $\sqsubseteq$, and *false* $\sqsupseteq t$.

(3) $t = \perp$.

From this definition, it is easy to check that the equality function $==^{\diagup}$ satisfies the

conditions required to a constraint domain $\mathcal{D}$ for the $CFLP(\mathcal{D})$ scheme:

(1) **Polarity**: $t_1 ==^{\curlywedge} t_2 \to t$ behaves monotonically with respect to the arguments t_1 and t_2, and antimonotonically with respect to the result t. Formally, for all $t_1, t_1', t_2, t_2', t, t' \in D_{\curlywedge}$ such that $t_1 ==^{\curlywedge} t_2 \to t$, $t_1 \sqsubseteq t_1'$, $t_2 \sqsubseteq t_2'$, and $t \sqsupseteq t'$, $t_1' ==^{\curlywedge} t_2' \to t'$ also holds.

(2) **Radicality**: As soon as the arguments given to $==^{\curlywedge}$ have enough information to return a result, the same arguments suffice already for returning a total result. Formally, for all $t_1, t_2, t \in D_{\curlywedge}$, if $t_1 ==^{\curlywedge} t_2 \to t$ then $t = \bot$ or else there is some total $t' \in D_{\curlywedge}$ such that $t_1 ==^{\curlywedge} t_2 \to t'$ and $t' \sqsupseteq t$.

An *equality constraint* (or simply, $\curlywedge$*-constraint*) is a multiset $\{\!\{s, t\}\!\}$, written $s == t$, where $s, t \in \mathcal{T}(\mathcal{F}_\bot, \mathcal{V})$ are λ-terms of the same type. The set of *solutions* of an equality constraint $s == t$ is defined as follows: $Soln(s == t) = \{\gamma \in Subst(\mathcal{F}_\bot, \mathcal{V}) \mid t\gamma ==^{\curlywedge} s\gamma \to true\}$. Any set E of strict equations is interpreted as conjunction, and therefore $Soln(E) = \bigcap_{(s==t)\in E} Soln(s == t)$.

3.3 The $\curlywedge$-constraint solver

Solving equality constraints in first-order term algebras (which is also known as *unification*) is the most famous symbolic constraint solving problem. In the higher-order case, *higher-order unification* is a powerful method for solving equality $\curlywedge$-constraints between λ-terms and is currently used in *theorem provers* [11,12]. Other applications of higher-order unification include *program synthesis* and *machine learning* [13]. However, one of the major obstacles for reasoning in the higher-order case is that unification is undecidable. However, in this subsection we examine a decidable higher-order unification case of patterns by means of the development of a $\curlywedge$-constraint solver for the higher-order constraint domain $\curlywedge$, now supporting an improved treatment of the strict equality $==$ as a built-in primitive function symbol, rather than a defined function [3].

Definition 3.2 [States] The constraint solver $Solver^{\curlywedge}$ for the higher-order domain $\curlywedge$ acts on *states* of the form $P \equiv \langle E \,|\, \mathcal{K}\rangle$, where E is a set of strict equality constraints $s == t$ between λ-terms s, t, and $\mathcal{K}$ is a set of patterns intended to represent and store computed *values* in the sense of [15,16] during the constraint solving process. The meaning of a state $P \equiv \langle E \,|\, \mathcal{K}\rangle$ is as follows: $[\![\langle E \,|\, \mathcal{K}\rangle]\!] = \{\gamma \in Soln(E) \mid \mathcal{K}\gamma \text{ is a set of values }\}$. We note that $[\![\langle E \,|\, \mathcal{K}\rangle]\!] = \emptyset$ whenever $\mathcal{K}$ is not a set of values. In the sequel, we denote this state by `fail` and call it *failure state*.

Solving a set of strict equality $\curlywedge$-constraints amounts to computing $\curlywedge$*-derivations*, i.e., sequences of transformation steps.

Definition 3.3 [Derivations] A $\curlywedge$*-derivation* of a set E of strict equality $\curlywedge$-constraints is a maximal finite sequence of transformation steps: $P_0 \equiv \langle E \,|\, \emptyset\rangle \equiv \langle E_0 \,|\, \mathcal{K}_0\rangle \Rightarrow_{\sigma_1} P_1 \equiv \langle E_1 \,|\, \mathcal{K}_1\rangle \Rightarrow_{\sigma_2} \cdots \Rightarrow_{\sigma_m} P_m \equiv \langle E_m \,|\, \mathcal{K}_m\rangle$, between states P_0, P_1, …, P_m, such that $P_m \neq$ `fail` is a *final state*, i.e., a non failure state which can not be transformed anymore.

Definition 3.4 [λ-constraint solver]

(1) Each transformation step in a λ-derivation Π corresponds to an instance of some transformation rule of the λ-constraint solver $Solver^{\lambda}$ described below. We abbreviate Π by $P_0 \Rightarrow^*_\sigma P_m$, where $\sigma = \sigma_1 \ldots \sigma_m$.

(2) Given such a set E of strict equality λ-constraints, the set of *computed answers* produced by the λ-constraint solver $Solver^{\lambda}$ is $\mathcal{A}(E) = \{\, \sigma\gamma \restriction_{\mathcal{FV}(E)} \mid \langle E \,|\, \emptyset\rangle \Rightarrow^*_\sigma P$ is a λ-derivation and $\gamma \in [\![P]\!] \,\}$, where $\mathcal{FV}(E)$ is the set of *free variables* of E.

In the sequel, we will describe the transformation rules of the λ-constraint solver and analyze its main properties.

(an) **annotation**
$\langle \{\!\{ s == t, E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_{\{\}} \langle \{\!\{ s ==_H t, E \}\!\} \mid \mathcal{K} \cup \{H\}\rangle$
where H is a fresh variable of a suitable type.

(sg) **strict guess**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) ==_H t, E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) ==_{H\sigma} t, E \}\!\} \mid \mathcal{K}\sigma\rangle$
where $a \in \mathcal{F} \cup \{\overline{x_k}\}$, and $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$.

(d) **decomposition**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) ==_u \lambda\overline{x_k}.a(\overline{t_n}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ \overline{\lambda\overline{x_k}.s_n ==_{H_n} \lambda\overline{x_k}.t_n}, E \}\!\} \mid \mathcal{K}\sigma\rangle$
where $a \in \mathcal{F} \cup \{\overline{x_k}\}$, and either
- $u \equiv H$ and $\sigma = \{H \mapsto \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})\}$, or
- $u \equiv \lambda\overline{x_k}.a(\overline{H_n(\overline{x_k})})$ and $\sigma = \{\}$.

(i) **imitation**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) ==_u \lambda\overline{x_k}.f(\overline{t_n}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ \overline{\lambda\overline{x_k}.X_n(\overline{s_p}) ==_{H_n} \lambda\overline{x_k}.t_n}, E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma\rangle$
where $X \in \mathcal{V}$, and either
- $u \equiv H$ and $\sigma = \{X \mapsto \lambda\overline{y_p}.f(\overline{X_n(\overline{y_p})}), H \mapsto \lambda\overline{x_k}.f(\overline{H_n(\overline{x_k})})\}$, or
- $u \equiv \lambda\overline{x_k}.f(\overline{H_n(\overline{x_k})})$ and $\sigma = \{X \mapsto \lambda\overline{y_p}.f(\overline{X_n(\overline{y_p})})\}$.

(p) **projection**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) ==_u t, E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ \lambda\overline{x_k}.X(\overline{s_p}) ==_u t, E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma\rangle$
where $X \in \mathcal{V}$, t is not flex, and $\sigma = \{X \mapsto \lambda\overline{y_p}.y_i(\overline{X_n(\overline{y_p})})\}$.

(fs) **flex same**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{y_p}) ==_H \lambda\overline{x_k}.X(\overline{y'_p}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ E \}\!\}\sigma \mid (\mathcal{K} \cup \{X\})\sigma\rangle$
where $X \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_p})$ and $\lambda\overline{x_k}.X(\overline{y'_p})$ are patterns, $\sigma = \{X \mapsto \lambda\overline{y_p}.Z(\overline{z_q}),\ H \mapsto \lambda\overline{x_k}.Z(\overline{z_q})\}$ with $\{\overline{z_q}\} = \{y_i \mid y_i = y'_i,\ 1 \le i \le n\}$.

(fd) **flex different**
$\langle \{\!\{ \lambda\overline{x_k}.X(\overline{y_p}) ==_H \lambda\overline{x_k}.Y(\overline{y'_q}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_\sigma \langle \{\!\{ E \}\!\}\sigma \mid (\mathcal{K} \cup \{X, Y\})\sigma\rangle$
where $X, Y \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_p})$ and $\lambda\overline{x_k}.Y(\overline{y'_q})$ are patterns, $X \neq Y$, $\sigma = \{X \mapsto \lambda\overline{y_p}.Z(\overline{z_r}), Y \mapsto \lambda\overline{y'_q}.Z(\overline{z_r}), H \mapsto \lambda\overline{x_k}.Z(\overline{z_r})\}$ with $\{\overline{z_r}\} = \{\overline{y_p}\} \cap \{\overline{y'_q}\}$.

(cf) **clash failure**
$\langle \{\!\{ \lambda\overline{x_k}.a(\overline{s_n}) ==_u \lambda\overline{x_k}.a'(\overline{t_m}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_{\{\}} \texttt{fail}$
if $a, a' \in \mathcal{F}_c \cup \{\overline{x_k}\}$ (where the notation $\mathcal{F}_c$ will be explained in Section 5), and either (i) $a \neq a'$ or (ii) $hd(u) \notin \mathcal{V} \cup \{a, a'\}$.

(oc) **occur check**
$\langle \{\!\{ \lambda\overline{x_k}.s ==_u \lambda\overline{x_k}.X(\overline{y_n}), E \}\!\} \mid \mathcal{K}\rangle \Rightarrow_{\{\}} \texttt{fail}$
if $X \in \mathcal{V}$, $\lambda\overline{x_k}.X(\overline{y_n})$ is a flex pattern, $hd(\lambda\overline{x_k}.s) \neq X$ and $(\lambda\overline{x_k}.s)|_p = X(\overline{z_n})$, where $\overline{z_n}$ is a sequence of distinct bound variables and p is a maximal *safe position* of $\lambda\overline{x_k}.s$ (i.e., $hd((\lambda\overline{x_k}.s)|_q) \in \mathcal{BV}(\lambda\overline{x_k}.s, q) \cup \mathcal{F}_c$ for all $q \le p$).

In order to illustrate the overall behavior of our constraint solver $Solver^{\lambda}$, we consider the following λ-derivation involving the function symbols given in the signature of the *diff*-example presented in Section 1: $\langle \{\!\{ \lambda x.\, sin(F(x)) == \lambda x.\, sin(cos(x)) \}\!\} \mid \emptyset\rangle \Rightarrow^{(an),(d),(i)}_{\{F \mapsto \lambda x.\, cos(x)\}} \langle\, \emptyset \mid \{\lambda x.\, sin(cos(x)), \lambda x.\, cos(x)\}\rangle$. Therefore, we have computed the substitution $\{F \mapsto \lambda x.\, cos(x)\}$ as the only answer in $\mathcal{A}(\lambda x.\, sin(F(x)) ==$

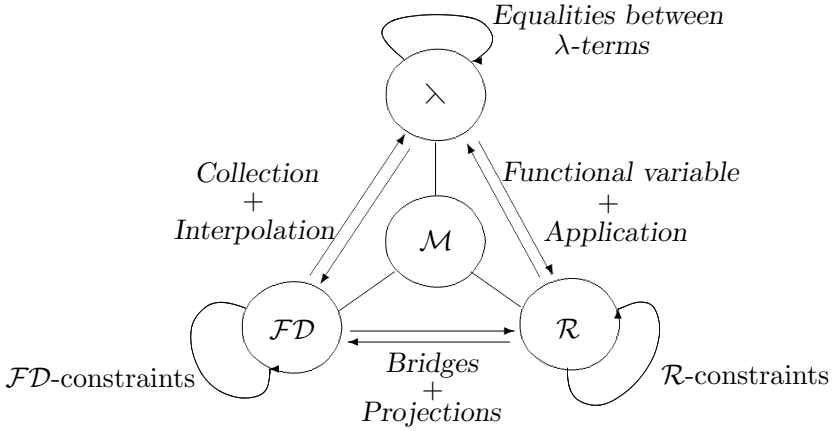

Fig. 2. The higher-order coordination domain $\mathcal{C} = \mathcal{M} \oplus \diagdown \oplus \mathcal{FD} \oplus \mathcal{R}$.

$\lambda x.\, sin(cos(x)))$.

The main properties of the $\diagdown$-constraint solver, *soundness* and *completeness*, relate the solutions of a set of strict equality $\diagdown$-constraints to the answers computed by our system of transformation rules for higher-order unification.

Theorem 3.5 (Properties of the $\diagdown$-constraint solver)

(1) **Soundness***: Let $\langle E \mid \emptyset \rangle \Rightarrow_\sigma^* P$ be a $\diagdown$-derivation. Then, $\sigma\gamma \in Soln(E)$ whenever $\gamma \in [\![P]\!]$.*

(2) **Completeness***: Let E be a set of $\diagdown$-constraints. Then, $\mathcal{A}(E) = \{\gamma\restriction_{\mathcal{FV}(E)} \mid \gamma \in Soln(E)\}$.*

4 Higher-Order Coordination of Algebraic Domains

The higher-order domain $\diagdown$ supports computations with symbolic equality $\diagdown$-constraints over λ-abstractions involving values of arbitrary user-defined datatypes. However, from a programmer's viewpoint we also need to work with extended algebraic constraint domains supporting computations with arithmetic and equality $\diagdown$-constraints over λ-terms involving numerical values. For this reason, in the context of our higher-order *CFLP* framework, we need to introduce extensions of both the classical domain $\mathcal{R}$, which supplies arithmetic constraints over real numbers, and the finite domain $\mathcal{FD}$, which supplies arithmetic and finite domain constraints over integers, to deal now with λ-abstractions defined over real numbers and integers.

A convenient formal definition of the algebraic constraint domain $\mathcal{R}$ is as follows: $\mathcal{R}$ is a structure $\langle D_\mathcal{R}, \{p^\mathcal{R}\}\rangle$, where the set of base values includes just one base type *real* whose values represent real numbers, the carrier set $D_\mathcal{R}$ coincides with the set of ground patterns over real numbers, and the usual interpretations $p^\mathcal{R}$ include as primitive function symbols the strict equality operator $==$, defined as for the $\diagdown$-domain, the arithmetical operators $+$, $-$, $*$, $/ :: real \to real \to real$, and the inequality operator $\leq :: real \to real \to bool$. Concerning the solver $solver^\mathcal{R}$, we expect that it is able to deal with $\mathcal{R}$-derivations of $\mathcal{R}$-specific constraints sets consisting of primitive constraints of the following two kinds: proper $\mathcal{R}$-constraints

involving an arithmetic operator, and specific higher-order $\mathcal{R}$-constraints having the form $t_1 == t_2$, where t_1 and t_2 are λ-terms over real constant values or variables whose type is known to be *real* prior to the solver invocation. We assume that $Solver^{\mathcal{R}}$ is implemented as a *black-box* solver on top of *SICStus Prolog*, and solves $\mathcal{R}$-specific higher-order constraints in a way compatible with the behavior of the $\nwarrow$-solver described in the previous section.

Analogously, $\mathcal{FD}$ is a structure $\langle D_{\mathcal{FD}}, \{p^{\mathcal{FD}}\}\rangle$, where the set of base values includes just one base type *int* whose values represent integer numbers, the carrier set $D_{\mathcal{FD}}$ coincides with the set of ground patterns over integer numbers, and the usual interpretations $p^{\mathcal{FD}}$ include as primitive function symbols the strict equality operator $==$, the arithmetical operators $+$, $-$, $*$, $/ :: int \to int \to int$, the inequality operator $\leq :: int \to int \to bool$, and the following primitive function symbols (see [7] for more details):

- $domain :: [int] \to int \to int \to bool$, to fix that a non-empty list of integer variables belongs to an interval of values.
- $labeling :: [labelType] \to [int] \to bool$, to select an integer variable of the list with a non-empty, non-singleton domain, selecting a value of this domain and assigning the value to the variable, where *labelType* is an enumerated datatype used to represent *labeling strategies* [7].

Concerning the solver $solver^{\mathcal{FD}}$, we also assume that is implemented as a *black-box* solver on *SICStus Prolog*, and we expect that it is able to deal with $\mathcal{FD}$-derivations of proper $\mathcal{FD}$-constraints involving arithmetic $\mathcal{FD}$-operators and higher-order algebraic constraints $t_1 == t_2$, where t_1 and t_2 are λ-terms over integer constant values or variables whose type is known to be *int* prior to the solver invocation.

A *coordination domain* $\mathcal{C}$ is a kind of "hybrid" constraint domain built from various component domains (as e.g., $\mathcal{H}, \nwarrow, \mathcal{R}, \mathcal{FD}, \ldots$) intended to cooperate. The construction of coordination domains involves a so-called *mediatorial domain* $\mathcal{M}$, whose purpose is to supply mechanisms for communication among the component domains via bridges, projections, functional variable applications, interpolations, and some more ad hoc operations (see **Fig. 2**). In this work, the component domains will be chosen as the pure domains $\nwarrow$, $\mathcal{R}$, and $\mathcal{FD}$, equipped with constraint solvers, in such a way that the communication provided by the mediatorial domain will also benefit the solvers. In the remaining of this section we briefly explain the construction of this *higher-order coordination domain* $\mathcal{C}$, represented as the sum $\mathcal{C} = \mathcal{M} \oplus \nwarrow \oplus \mathcal{FD} \oplus \mathcal{R}$.

Mathematically, the construction of the coordination domain $\mathcal{C}$ relies on a combined algebraic constraint domain $\mathcal{FD} \oplus \mathcal{R}$, which represents the *amalgamated sum* of the two joinable algebraic domains $\mathcal{FD}$ and $\mathcal{R}$. In this case, the *joinability condition* asserts that the only primitive function symbol allowed to belong to $\mathcal{FD}$ and $\mathcal{R}$ is the strict equality $==$, where the interpretation of this operator will behave as defined for each algebraic constraint domain and $\nwarrow$. As a consequence, the amalgamated sum $\nwarrow \oplus \mathcal{FD} \oplus \mathcal{R}$ is always possible, and gives rise to compound a higher-order algebraic domain that can profit from the higher-order $\nwarrow$-constraint solver. However, in order to construct a more interesting sum for higher-order al-

gebraic cooperation tailored to the communication among pure domains $\diagdown$, $\mathcal{R}$, and $\mathcal{FD}$, *mediatorial domains* are needed.

The *higher-order mediatorial domain* $\mathcal{M}$ serves as a basis for useful cooperation facilities among $\diagdown$, $\mathcal{FD}$, and $\mathcal{R}$, including the projection of $\mathcal{R}$-constraints to the $\mathcal{FD}$-solver (and vice versa) using bridges, the specialization of $\diagdown$-constraints to become $\mathcal{R}$- or $\mathcal{FD}$-constraints, the definition of algebraic constraints in $\mathcal{R}$ and $\mathcal{FD}$ from the application of higher-order functional variables in the domain $\diagdown$, and the gathering of numeric data values to construct a λ-abstraction in $\diagdown$ which closely fits the data points by means of interpolation.

- More precisely, *bridge constraints* $X \rightleftharpoons RX$, with $\rightleftharpoons :: \mathit{int} \rightarrow \mathit{real} \rightarrow \mathit{bool}$, can be used either for *binding* or *projection* purposes. Binding in $\mathit{solver}^{\mathcal{M}}$ simply instantiates a variable occurring at one end of a bridge whenever the other end of the bridge becomes a numeric value. Projection is a more complex operation which infers constraints to be placed in $\mathcal{R}$'s store from the constraints available in $\mathcal{FD}$'s store (and vice versa) and the relevant bridges available in $\mathcal{M}$. This enables each solver to take advantage of the computations performed by other solvers. We postulate a projection function $\mathit{proj}^{\mathcal{FD}\rightarrow\mathcal{R}}$ such that for any set $C_{\mathcal{FD}}$ of $\mathcal{FD}$-constraints and any finite set M of bridge constraints, $\mathit{proj}^{\mathcal{FD}\rightarrow\mathcal{R}}(C_{\mathcal{FD}}, M)$ returns a finite disjunction $C_{\mathcal{R}}$ of equivalent $\mathcal{R}$-constraints (similarly, $\mathit{proj}^{\mathcal{R}\rightarrow\mathcal{FD}}$). In order to maximize the opportunities for projection, we postulate a function $\mathit{bridges}^{\mathcal{FD}\rightarrow\mathcal{R}}$ such that $\mathit{bridges}^{\mathcal{FD}\rightarrow\mathcal{R}}(C_{\mathcal{FD}}, M)$ returns a finite set of new bridge constraints M' from the new variables in $C_{\mathcal{FD}}$ (similarly, $\mathit{bridges}^{\mathcal{R}\rightarrow\mathcal{FD}}$).
- *Interpolation* is the process of defining a function that takes on specified values at specified points. We use this technique in our higher-order setting to support the cooperation and communication between an algebraic constraint domain ($\mathcal{R}$ or $\mathcal{FD}$) and $\diagdown$. We try to construct a function, represented by a λ-abstraction, which must go through the data points. In order to apply this technique, our cooperative computation model keeps a store C by means of the execution of an algebraic constraint *collect* [...] C (similar to the `setof` predicate in *Prolog*) from a finite list of real or integer variables. Then, we assume an interpolation function *interpolation*, interpreted with respect to a mapping $\mathit{interp}^{\mathcal{R}\rightarrow\diagdown}$ (similarly, $\mathit{interp}^{\mathcal{FD}\rightarrow\diagdown}$), such that the higher-order constraint *interpolation* [...] C F returns in the functional variable F a λ-abstraction $\lambda\overline{u}.\, F(\overline{u})$ according to a predefined list of *interpolation methods* (implemented in *C++* and called from *Prolog*), so that the following *interpolation condition* holds: $F(\overline{x}) = y$ for all $(\overline{x}, y) \in C$. For communicating information between the higher-order domain $\diagdown$ and an algebraic constraint domain $\mathcal{D}$ ($\mathcal{R}$ or $\mathcal{FD}$) we can assume a mapping $\mathit{apply}^{\diagdown\rightarrow\mathcal{D}}$, defined by means of the application of the functional variable F associated to a λ-abstraction $\lambda\overline{u}.\, F(\overline{u})$ to a type appropriate finite number of arguments $\overline{x}$ in order to compound algebraic constraints from $F(\overline{x})$ (**Fig. 3** illustrates this cooperation of the algebraic domains $\mathcal{FD}$ and $\mathcal{R}$ with $\diagdown$ for the motivating examples presented in Section 2).

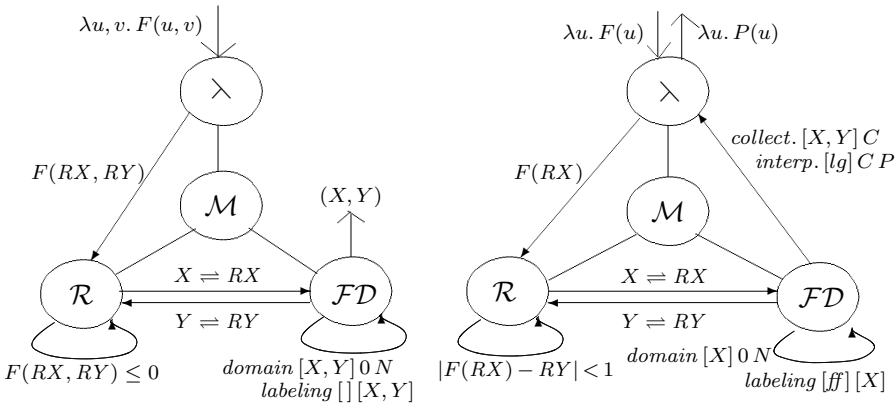

Fig. 3. Examples of higher-order programming with algebraic constraint cooperation.

5 Higher-Order Cooperative Programming in $CFLP(\mathcal{C})$

We are now ready to present our computation framework for higher-order functional and logic programming with cooperation of algebraic constraint domains within the $CFLP(\mathcal{C})$ instance of the *CFLP* scheme. We sketch a prototype implementation of the $CFLP(\mathcal{C})$ computational model on top of the $\mathcal{TOY}$ system [6]. **Fig. 4** shows the architectural components of the higher-order cooperation schema in this system. The higher-order constraint domain $\diagdown$ and the algebraic domains $\mathcal{R}$ and $\mathcal{FD}$ with a mediatorial domain $\mathcal{M}$ to yield the coordination domain $\mathcal{C} = \mathcal{M} \oplus \diagdown \oplus \mathcal{FD} \oplus \mathcal{R}$ are supported by this implementation. The main novelty here is that compilation proceeds by performing a translation from higher-order programs on the domain $\diagdown$ to typed higher-order applicative programs in the Herbrand domain $\mathcal{H}$. Following [16], the idea consists in introducing an explicit application operation *apply*, replacing λ-abstractions (and similar constructs in our higher-order setting, such as partial applications) by means of new data constructors, and providing rewrite rules to define the proper behavior of the application operation when meeting terms where these new data constructors appear. Proper $\mathcal{FD}$ and $\mathcal{R}$ constraints, as well as $\diagdown$ and $\mathcal{H}$ constraints specific to $\mathcal{FD}$ and $\mathcal{R}$ are posted to the respective stores and handled by the respective *SICStus Prolog* solvers. On the other hand, the stores and solvers for the domains $\diagdown$, $\mathcal{H}$, and $\mathcal{M}$ are built into the code of the $\mathcal{TOY}$ implementation, rather than being provided by the underlying *SICStus Prolog* system. Moreover, the implementation of the fundamental mechanisms for algebraic domain cooperation: bridges, projections, and the collection and interpolation of data values among the higher-order domain $\diagdown$ and the algebraic domains $\mathcal{R}$ and $\mathcal{FD}$ are tackled by *glue code* integrating *Prolog* services with *C++* and using a so-called *mixed store* which keeps a representation of the mediatorial constraint store as one single *Prolog* structure.

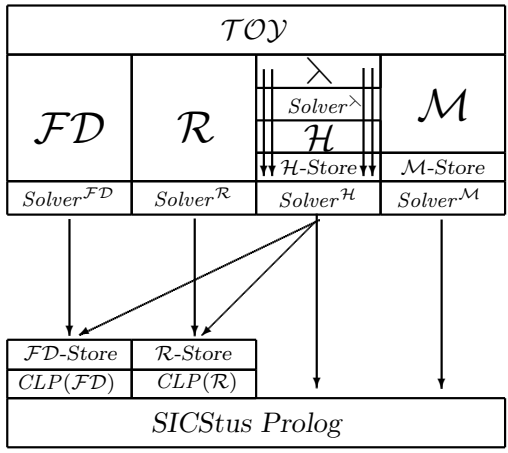

Fig. 4. Architectural components of the higher-order cooperation in $\mathcal{TOY}$.

6 Conclusions

In this work we have presented an efficient use of cooperative algebraic constraint domains and solvers in a higher-order functional and logic programming framework on λ-abstractions. We have investigated foundational and practical issues concerning a sound and complete computational framework for the cooperation of algebraic constraint domains. We have designed an improved higher-order instance $CFLP(\mathcal{C})$ of an already existing generic scheme $CFLP(\mathcal{D})$ for constraint functional logic programming over a higher-order coordination domain $\mathcal{C}$, as well as a prototype implementation in the $\mathcal{TOY}$ system which supports the cooperation via bridges, projections, application of functional variables, and polynomial interpolations on λ-abstractions.

In addition to already mentioned works, an important related work in this area is the *CFLP* scheme developed by Mircea Marin in his PhD Thesis [8]. This work introduces $CFLP(\mathcal{D}, \mathcal{S}, \mathcal{L})$, a family of languages parameterized by a constraint domain $\mathcal{D}$, a strategy $\mathcal{S}$ which defines the cooperation of several constraint solvers over $\mathcal{D}$, and a constraint lazy narrowing calculus $\mathcal{L}$ for solving constraints involving functions defined by user given constrained rewrite rules. The main difference with respect to our approach is the lack of declarative (model-theoretic and fixpoint) semantics provided by the rewriting logic underlying our $CFLP(\mathcal{C})$ instance (see [16] for more details). Another difference with respect to our approach is the intended application domain. The instance of *CFLP* developed by Marin combines four solvers over a constraint domain for algebraic symbolic computation.

In the future, we would like to improve some of the limitations of our current approach to higher-order algebraic domain cooperation, concerning both the formal foundations and the implemented system. For instance, the computational model should be generalized to allow for an arbitrary higher-order coordination domain $\mathcal{C}$ in place of the concrete choice $\mathcal{M} \oplus \nwarrow \oplus \mathcal{R} \oplus \mathcal{FD}$, and the implemented prototype should be properly developed, maintained and improved in various ways. In particular, the experimentation with *benchmarks* and application cases should be further

developed.

References

[1] S. Estévez, T. Hortalá, M. Rodríguez, R. del Vado et al. *On the cooperation of the constraint domains $\mathcal{H}$, $\mathcal{R}$, and $\mathcal{FD}$ in CFLP*. Journal of *Theory and Practice of Logic Programming*, vol. 9, pp. 415–527, 2009.

[2] J.C. González, M.T. Hortalá, and M. Rodríguez. *A higher-order rewriting logic for functional logic programming*. In Proc. *ICLP'97*, pp. 153–167, 1997.

[3] M. Hanus. *The Integration of Functions into Logic Programming: From Theory to Practice*. Journal of *Logic Programming* 19&20, pp. 583–628, 1994.

[4] M. Hanus and C. Prehofer. *Higher-order narrowing with definitional trees*. *Journal of Functional Programming*, vol. 9, pp. 33–75, 1999.

[5] J.R. Hindley and J.P. Seldin. Introduction to Combinatorics and λ-Calculus. *Cambridge University Press*, 1986.

[6] F.J. López and J. Sánchez. $\mathcal{TOY}$: A Multiparadigm Declarative System. In Proc. *RTA'99*, Springer LNCS 1631, pp 244–247, 1999. System and documentation available at `http://toy.sourceforge.net`.

[7] F.J. López, M. Rodríguez, and R. del Vado. *A New Generic Scheme for Functional Logic Programming with Constraints*. Journal of *Higher-Order and Symbolic Computation*, vol. 20 (1/2), pp. 73–122, 2007.

[8] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD. Thesis, Johannes Kepler Universität Linz, 2000.

[9] D. Miller. *A logic programming language with λ-abstraction, function variables, and simple unification*. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[10] G. Nadathur and D. Miller. *An overview of λ-Prolog*. In *Proc. Int. Conf. on Logic Programming (ICLP'88)*, The MIT Press, pp. 810–827, 1988.

[11] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Springer LNCS*, vol. 2283, 2002.

[12] L.C. Paulson. *Isabelle: A Generic Theorem Prover*. *Springer LNCS*, vol. 828, 1994.

[13] C. Prehofer. *Solving Higher-Order Equations. From Logic to Programming*. Foundations of Computing. Birkäuser Boston, 1998.

[14] T. Suzuki, K. Nakagawa, and T. Ida. *Higher-Order Lazy Narrowing Calculus: A Computation Model for a Higher-Order Functional Logic Language*. In Proc. of *ALP'97*, vol. 1298 of LNCS, pp. 99–113, 1997.

[15] R. del Vado. *A Higher-Order Demand-Driven Narrowing Calculus with Definitional Trees*. In Proc. *ICTAC'07*, Springer LNCS, vol. 4711, pp. 169–184, 2007.

[16] R. del Vado. *A Higher-Order Logical Framework for the Algorithmic Debugging and Verification of Declarative Programs*. In *PPDP'09*, ACM, pp. 49–60, 2009.