

# Playing with $\mathcal{TOY}$ : Constraints and Domain Cooperation

Sonia Estévez-Martín<sup>1</sup>, Antonio J. Fernández<sup>2</sup>, and Fernando Sáenz-Pérez<sup>3,\*</sup>

<sup>1</sup> Univ. Complutense de Madrid, Dpto. de Sistemas Informáticos y Comp, Spain

<sup>2</sup> Univ. de Málaga. Dpto. de Lenguajes y Ciencias de la Computación, Spain

<sup>3</sup> Univ. Complutense de Madrid, Dpto. de Inteligencia Artificial e Ing. SW, Spain

**Abstract.** This paper describes  $\mathcal{TOY}$ , an implementation of a Constraint Functional Logic Programming scheme  $CFLP(C)$ , where  $C$  is a coordination domain involving the cooperation among several constraint domains  $D_1, \dots, D_n$  via a mediatorial domain  $M$ . This implementation follows a cooperative goal solving calculus for  $CFLP(C)$  based on lazy narrowing, invocation of solvers for each domain  $D_i$ , and projection operations for converting  $D_i$  constraints into  $D_j$  constraints with the aid of mediatorial constraints supplied by  $M$ . Mediatorial constraints allow solving programs that require constraints of different domains, and projection may improve performance, allowing certain solvers to profit from (the projected forms) of constraints originally intended for other solvers. As a relevant concrete instance of our  $CFLP(C)$ , we implemented the cooperation among Herbrand, real arithmetic and finite domain constraints, and the mediatorial constraints relate numeric variables belonging to the last two domains. These mediatorial constraints are the bridge `#== :: int -> real -> bool` (that evaluates to true if their arguments are equivalent -i.e., the real value is considered to represent the integer one- and false otherwise), and the antibridge `#/= :: int -> real -> bool` (with a countermeaning).

## 1 Introduction

$\mathcal{TOY}$  [1] is a constraint functional logic language and system, designed to support the main declarative programming styles and their combination. From <http://toy.sourceforge.net> the preferred distribution for  $\mathcal{TOY}$  can be downloaded. There are some possibilities: Choose either a binary distribution (a portable application that does not need installation) or a source-code distribution (which requires SICStus Prolog previously installed). Therefore, almost any platform can run  $\mathcal{TOY}$  (e.g., the system can be started as a Windows application or in a Linux console). It features a command interpreter for submitting goals and system commands. In addition, it has been connected to ACIDE [2],

---

\* First and third authors were partially supported by the Spanish National Projects MERIT-FORMS (TIN2005-09027-C03-03) and PROMESAS-CAM(S-0505/TIC/0407). Second author was partially supported by Spanish MCyT projects under contracts TIN2005-08818-C04-01 and TIN-2007-67134.

a graphical and configurable integrated development environment. Further developments are also guided to port the system to free Prolog interpreters such as B-Prolog.

Programs in  $\mathcal{TOY}$  can include definitions of types, operators, lazy functions in Haskell style, as well as definitions of predicates in Prolog style. A predicate is viewed as a particular kind of function whose right-hand side is true. A function definition consists of an optional *type declaration* and one or more *defining rules*, which are possibly conditional rewrite rules. Both functions and predicates must be well-typed with respect to a polymorphic type system [3].

Programs can use constraints within the definitions of both predicates and functions. Constraints supported by the system include symbolic equations and disequations [4], linear and non-linear arithmetic constraints over the real numbers [5] and finite domain constraints [6].

$\mathcal{TOY}$  computations solve goals and display computed answers.  $\mathcal{TOY}$  solves goals by means of a demand driven lazy narrowing strategy [7] combined with constraint solving. Answer constraints can represent bindings for logic variables, as in answers computed by a Prolog system. Some features of  $\mathcal{TOY}$  are:

1. *Curried style*. This allows that partial applications of curried functions can be used to express functional values as partial patterns.
2. *Non-deterministic functions*. These are defined either by means of defining rules with overlapping left-hand sides or using extra variables in the right-hand side that do not occur in the left-hand side.
3. *Sharing* for values of all variables which occur in the left-hand sides of defining rules and have multiple occurrences in the right-hand side and/or the conditions. Sharing implements so-called *call-time choice* semantics of non-deterministic functions.
4. *Higher-order functions* in the style of Haskell, except that lambda abstractions are not allowed. In  $\mathcal{TOY}$ , higher-order can be naturally combined with non-determinism.
5. *Dynamic Cut*. Optimization that detects deterministic functions at compile time, and the generated code includes a test for detecting at run-time the computations that can actually be pruned [8].
6. *Finite Failure*. The primitive Boolean function *fails* is a direct counterpart to finite failure in Prolog.

## 2 Constraint Functional Logic Programming Scheme $CFLP(C)$

$\mathcal{TOY}$  implements a Constraint Functional Logic Programming scheme  $CFLP(D)$  over a parametrically given constraint domain  $D$ , proposed in [9].  $CFLP(D)$  is a logical and semantic framework for lazy Constraint Functional Logic Programming over  $D$ , which provides a clean and rigorous declarative semantics for  $CFLP$  languages.

In particular,  $D$  is the *coordination domain*  $C$  introduced in [10] as the amalgamated sums of the domains to be coordinated,  $D_1, \dots, D_n$ , along with a

*mediatorial domain*  $M$  which supplies special communication constraints, called bridges, used to impose the equivalence between values of different base types.

The Cooperative Constrained Lazy Narrowing Calculus  $CCLNC(C)$  presented in [10] provides a fully sound formal framework for functional logic programming with cooperating solvers over various constraint domains.  $CCLNC(C)$  has been proved fully sound w.r.t.  $CRWL(C)$  semantics [9].

### 3 Cooperation in $\mathcal{TOY}$ : Bridges and Projections

$\mathcal{TOY}$  comes equipped with solvers corresponding to three constraint domains:

1. *Herbrand*, with equality and disequality constraints.
2. *Real Arithmetic*, with arithmetical constraints over real numbers.
3. *Finite domain*, with constraints over integer numbers.

The Herbrand Solver is always available, and the real and finite domain solvers can be optionally loaded. With the aim of extending the system applicability, a mechanism for solver cooperation on these domains has been recently incorporated. This mechanism has two main pillars: *Bridges*, necessary for solver communication, and *Projections*, that improve the efficiency of some programs.

A bridge is a special kind of ‘hybrid’ constraint which allows the communication among the real and finite (‘pure’) domains and instantiates a variable occurring at one end of a bridge whenever the other end becomes a numeric value. Note that, a bridge constraint can be used to impose an integral constraint over its right argument. As an example, suppose we want to know if two different lines can meet at one integer point. A line can be described algebraically by the linear equation  $y = m * x + b$ , and the corresponding  $\mathcal{TOY}$  program is:

Program	Goal	Answer
<pre> meetLines M1 B1 M2 B2=(X,Y) &lt;== X #== RX,       Y #== RY,       RY == M1*RX + B1,       RY == M2*RX + B2 </pre>	<pre> meetLines 2 4 1 2 == L meetLines 1 1 1 2 == L meetLines 1 1 3 2 == L </pre>	<pre> L -&gt; (-2, 0) no %parallel no %real point </pre>

Projection takes place during goal solving whenever a pure constraint is submitted to its solver. At that moment, projection builds a mate constraint which is submitted to the mate solver (think of finite domain solver as the mate of real solver, and vice versa). Projection rules described in [10,11] relying on the available bridges are used for building mate constraints. For example, suppose we want to calculate the intersection of a triangular region (defined in the continuous plane) with an  $(N \times N)$ -size square discrete grid (defined in the discrete plane). A  $\mathcal{TOY}$  program that solves the problem, for any given even integer number  $N$ , is shown below; the triangular region is described by the inequalities whereas the square grid is described by the finite domain constraints (i.e., those labelled with  $\#$  and the function *labeling/2*).

Program	Mate Constraints
<code>bothIn L X Y N :- X#==RX, Y#==RY, N#==NX,</code>	
<code>RY &gt;= (NX/2) - 0.5,</code>	$\Rightarrow Y \#>= \lceil NX/2 \# - 0.5 \rceil,$
<code>RY - RX &lt;= 0.5,</code>	$\Rightarrow Y \#- X \#<= \lfloor 0.5 \rfloor,$
<code>RY + RX &lt;= NX + 0.5,</code>	$\Rightarrow Y \#+ X \#<= \lfloor NX \#+ 0.5 \rfloor,$
<code>domain [X,Y] 0 N,</code>	$\Rightarrow 0 \leq RX, RX \leq N, 0 \leq RY, RY \leq N$
<code>labeling L [X,Y]</code>	

Mate constraints, generated during goal solving, allow the finite domain solver to drastically prune the domains of  $X$  and  $Y$ . Therefore, if we have a huge grid and a tiny triangle and the projection is enabled, then the computation time is drastically reduced. Note that not all the constraints are projected, for example the labeling constraint.

We have borrowed the idea of constraint projection from the work of P. Hofstedt [12], adapting it to our *CFLP* scheme and adding bridge constraints as a novel technique which makes projections more flexible and compatible with type discipline.

## References

1. Arenas, P., Fernández, A., Gil, A., López, F., Rodríguez, M., Sáenz, F.:  $\mathcal{TOY}$ . In: Caballero, R., Sánchez, J. (eds.) A Multiparadigm Declarative Language. Version 2.3.0 (2007), Available at <http://toy.sourceforge.net>
2. Sáenz-Pérez, F.: ACIDE: An Integrated Development Environment Configurable for LaTeX. The *PracTeX Journal* 2007(3) (2007)
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: *POPL* 1982, pp. 207–212. ACM Press, New York (1982)
4. Arenas, P., Gil, A., López, F.: Combining Lazy Narrowing with Disequality Constraints. In: Penjam, J. (ed.) *PLILP* 1994. LNCS, vol. 844, pp. 385–399. Springer, Heidelberg (1994)
5. Hortalá, T., López, F., Sánchez, J., Ullán, E.: Declarative Programming with Real Constraints. Research Report SIP 5997, U.C.M (1997)
6. Fernández, A.J., Hortalá, T., Sáenz, F., del Vado, R.: Constraint Functional Logic Programming over Finite Domains. *Theory Pract. Log. Program.* 7(5), 537–582 (2007)
7. Loogen, R., López-Fraguas, F.J., Rodríguez-Artalejo, M.: A demand driven computation strategy for lazy narrowing. In: Penjam, J., Bruynooghe, M. (eds.) *PLILP* 1993. LNCS, vol. 714, pp. 184–200. Springer, Heidelberg (1993)
8. Caballero, R., García-Ruiz, Y.: Implementing dynamic cut in toy. *Electr. Notes Theor. Comput. Sci.* 177, 153–168 (2007)
9. López, F., Rodríguez, M., del Vado, R.: A new generic scheme for functional logic programming with constraints. *Higher-Order and Symbolic Computation* 20(1/2), 73–122 (2007)
10. Estévez, S., Fernández, A.J., Hortalá, M.T., Rodríguez, M., del Vado, R.: A fully sound goal solving calculus for the cooperation of solvers in the CFLP scheme. *ENTCS* 177, 235–252 (2007)
11. Estévez, S., Fernández, A., Hortalá, T., Rodríguez, M., Sáenz, F., del Vado, R.: A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming. *ENTCS* 188, 37–51 (2007)
12. Hofstedt, P., Pepper, P.: Integration of declarative and constraint programming. *Theory Pract. Log. Program.* 7(1-2), 93–121 (2007)