About Implementing a Constraint Functional Logic Programming System with Solver Cooperation

Sonia Estévez-Martín[1], Antonio. J. Fernández[2] and Fernando Sáenz-Pérez[3]

[1] Univ. Complutense de Madrid, Dpto. de Sistemas Informáticos y Comp, Spain
[2] Univ. de Málaga. Dpto. de Lenguajes y Ciencias de la Computación, Spain
[3] Univ. Complutense de Madrid, Dpto. de Inteligencia Artificial e Ing. SW, Spain

Abstract. This paper provides unreported implementation details of a programming system which implements a seamless integration of constraint, functional, and logic paradigms, and that recently has incorporated a mechanism for solver cooperation on several domains: Herbrand (with equality and disequality constraints), finite domains (for constraint programming constraints over integers), and real numbers (for linear and non-lineal constraints). In particular, the cooperation of constraint solvers over numerical domains is specially interesting because of their practical use for developing many heterogeneous applications relating variables in both domains. This paper gives information about the compilation scheme of the system, its specific libraries, and focuses particularly on how solver cooperation has been integrated into the system. Regarding this last issue, the paper also provides preliminary performance results that supports the suitability of this cooperation mechanism.

1 Introduction

The addition of constraint solving technology to declarative programming systems has caused that these systems are now being considered suitable options (i.e., alternatives to traditional programming systems such as the imperative programming-based systems) for programming complex and real problems. Existing declarative constraint languages are high level programming tools that ease the task of programming (wrt. the formulation of the problem and/or program analysis) and provide a reasonable balance between program formulation and solving efficiency. Moreover, declarative constraint systems combine a high level of abstraction and a declarative nature with an extreme flexibility in the design of their implementations (e.g., wrt. their execution model). This means that they can be used not only as development tools for implementing non-trivial applications but also as platforms where research on key concepts of the implementation of programming languages (including concurrent/parallel models and memory management) can be done.

In this context, the design, implementation, and optimization of declarative constraint programming systems can be considered one of the major issues

treated in recent years in the constraint programming and logic programming areas. In this paper, we consider specifically the constraint functional logic programming (CFLP) language $\mathcal{TOY}$ [1, 2]. This language combines functional and relational notation, curried expressions, higher-order functions, patterns, partial applications, non-determinism, lazy evaluation, logical variables, types, domain variables, and constraint composition. It also provides technology for finite domain ($\mathcal{FD}$) constraint solving (including a wide set of $\mathcal{FD}$ constraints comparable to existing CLP($\mathcal{FD}$) systems and which is competitive with them as shown by performance results [3]), support for managing arithmetic linear and non-linear constraints defined on the real domain $\mathcal{R}$ [4], and provision of strict equality and disequality constraints [5] defined in the Herbrand domain $\mathcal{H}$. Each domain-specific constraint is solved in the associated domain-specific solver (i.e., $solve^{\mathcal{FD}}$, $solve^{\mathcal{R}}$, $solve^{\mathcal{H}}$, respectively) that are connected to the system via an adequate interface.

The set of constraint solvers of $\mathcal{TOY}$ provides support for solving a wide set of practical problems that require constraint solving over each single domain. However, there exist many practical problems that are better expressed using heterogeneous constraints (i.e., involving more than one domain) and, as a consequence, the formulation of these practical problems has to be artificially adapted to one of the domains supported by the connected solvers. With the aim of extending the applicability of the system, $\mathcal{TOY}$ has incorporated recently new features such as solver cooperation. The implementation of this feature in $\mathcal{TOY}$ is based on the theoretical framework described in [6].

This paper focuses specifically on implementation issues, not reported so far, of the $\mathcal{TOY}$ system. Among these issues, the paper briefly describes the compilation procedure and, more particularly, how solver cooperation, as described in [6], has been implemented. Thus, this paper can help other implementors of declarative constraint systems to understand the implementation fundamentals of $\mathcal{TOY}$, and can provide them further ideas to incorporate in their systems. In addition, some performance results are given to show the effectiveness of the solver cooperation mechanism implemented in $\mathcal{TOY}$.

2 Compiling Programs

$\mathcal{TOY}$ programs consist of *datatypes*, *type alias*, *infix operator* definitions, and rules for defining *functions*. The syntax is mostly borrowed from Haskell with the remarkable exception that variables and type variables begin with uppercase letters, whereas constructor symbols and type symbols begin with lowercase (see Example in Section 4.3). In particular, functions are *curried* and the usual conventions about associativity of application hold. As usual in functional programming, types are inferred, checked and, optionally, can be declared in the program.

Instead of using an abstract machine for running byte-code or intermediate code from compiled programs, the $\mathcal{TOY}$ system relies on an efficient Prolog system (i.e., SICStus Prolog [7]) for running $\mathcal{TOY}$ programs compiled to Pro-

log, as in other related systems [8]. The compilation follows a demand driven computation strategy for lazy narrowing [9], and its data-flow is described next.

A $\mathcal{TOY}$ program `program.toy` is compiled as follows (see Figure 1): First, functions, types and constructors defined in this user program are joined with predefined ones (coded in the file `basic.toy`). Next, lexical, syntactical, and semantical analysis are done, and the result contains type declarations, functional dependencies, and definitional trees [10]. Finally, from the last intermediate file, non-declared types are inferred and user-declared types are checked, generating the compiled Prolog code in `program.pl`. In addition, this result contains, among others, code for dynamic cut [11], totality constraints [12], type declarations for predefined functions and constructors, head normal form (hnf) computations, definitions for partial applications and declarations of precedence and associativity for infix operators. This Prolog code is compiled (from Prolog to the underlying SICStus abstract machine) and loaded into the SICStus system, in order to be able to evaluate expressions (i.e., solve goals) typed at the system prompt.

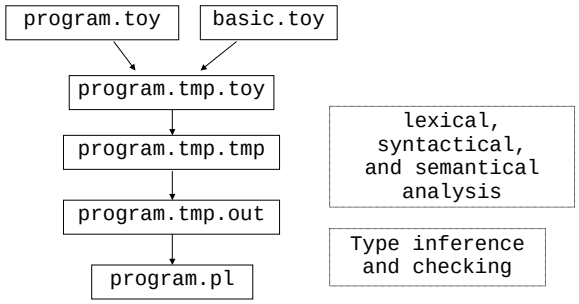

Fig. 1. Compilation Data-Flow

3 Libraries

$\mathcal{TOY}$ provides a number of independent libraries that contain specific definitions for types, data constructors and functions that enable an adequate handling of files, graphics, constraints over real numbers, and constraints over finite domains for integers (Herbrand constraints are always available). When these libraries are loaded, via the appropriate commands typed at the system prompt, these definitions are added to the basic ones. More specifically:

- **Files:** This library provides functions to handle text files, and includes operations to read from and write to files.
- **Graphics:** This library contains functions for building GUIs (Graphical User Interfaces) based on the Tcl/Tk library.

– **Constraints over Reals:** This library enables the constraint domain $\mathcal{R}$ with arithmetical constraints over real numbers, both linear and non-linear. However, these last ones are suspended until they become linear via instantiations. Note that loading this library implies that arithmetical (in)equations require no groundness on their variables.
– **Constraints over Finite Domains:** This library enables the constraint domain $\mathcal{FD}$ with finite domain constraints over integer numbers. For a detailed explanation of this kind of constrains see [1].

4 Implementing Solver Cooperation

This section describes the implementation fundamentals of the cooperation mechanism. Initially, an outline about the architectural components involved in the implementation of the cooperation mechanism is given, and a global overview of the two main pillars of this mechanism (i.e., bridges and projections) is provided. Further, an example of cooperation using both bridges and projections is shown. Later, the implementation of bridges and projections is described in detail. The section ends by giving some relevant comments about how constraint information is exchanged among solvers supported in $\mathcal{TOY}$, and by discussing related work.

4.1 Architectural Components of the Cooperation Schema

Figure 2 shows the Herbrand domain $\mathcal{H}$ for equality and disequality constraints dealing with constructed terms, $\mathcal{R}$ for (linear and non-linear) arithmetical constraints over real numbers, $\mathcal{FD}$ for finite domain constraints over integers, and the mediatorial constraint domain $\mathcal{M}$ for communication constraints among solvers, allowing their cooperation by means of *bridges* and *projections*. This last domain is a hybrid domain that supplies *bridge constraints* (`X #== Y`) for the communication among $\mathcal{H}$, $\mathcal{FD}$ and $\mathcal{R}$ domains (cf. Section 4.2).

Each constraint domain ($\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$, and $\mathcal{M}$) has an attached constraint store (*H*, *R*, *FD*, and *M*, resp.) and solver ($solve^{\mathcal{H}}$, $solve^{\mathcal{R}}$, $solve^{\mathcal{FD}}$, and $solve^{\mathcal{M}}$, resp.). We take advantage of the SICStus Prolog constraint stores for storing $\mathcal{R}$ and $\mathcal{FD}$ primitive constraints.

$\mathcal{TOY}$ provides lazy narrowing dealing with constraints and takes care of decomposing constraints by introducing new local (produced) variables [13]. Eventually, primitive constraints for $\mathcal{R}$ and $\mathcal{FD}$ arise, which must be submitted to their respective solvers i.e., $solve^{\mathcal{R}}$ and $solve^{\mathcal{FD}}$, resp., and stored in their corresponding stores. $\mathcal{TOY}$ uses the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog glue code for interfacing them with $solve^{\mathcal{FD}}$ and $solve^{\mathcal{R}}$, respectively, code for implementing $solve^{\mathcal{H}}$, and code for implementing lazy narrowing dealing with constraints. $solve^{\mathcal{M}}$ follows [6, 14] and the implementation of its bridge constraint is described next in Section 4.4.

Equality and disequality constraints for $\mathcal{H}$ are implemented as already reported in [5]. Also, disequality constraints may affect a variable whose type is

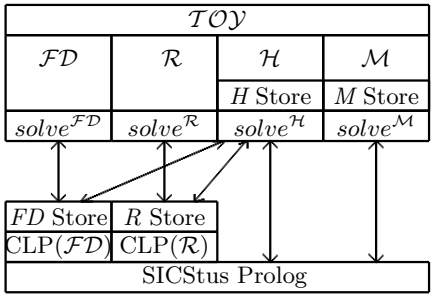

Fig. 2. Architectural Components of the Cooperation Schema in TOY

unknown [15]. These constraints are assumed to involve constructed terms and, therefore handled by $solve^{\mathcal{H}}$. But, along computation, these variables may be instantiated to a number, so that the corresponding disequality constraint is moved to $solve^{\mathcal{R}}$, or $solve^{\mathcal{FD}}$, depending on whether this number is a real or an integer, resp.

4.2 Bridges and Projections: Cooperation Fundamentals

Basically, the cooperation mechanism implemented in the system allows the communication among $\mathcal{H}$, $\mathcal{FD}$, and $\mathcal{R}$, using special communication constraints called *bridges*. Bridges implement binding, equivalence between numbers in those constraint domains, as well as disequalities (antibridges) between variables in those constraint stores. A bridge `X #== Y` constrains `X` $\in \mathbb{Z}$ and `Y` $\in \mathbb{R}$ to take the same integer value. Bridges are kept in a special store and they are used for two purposes, namely *binding* and *projection*. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end becomes a numeric value. Projection is a more complex operation which takes place whenever a pure constraint is submitted to $solve^{\mathcal{FD}}$ or $solve^{\mathcal{R}}$. At that moment, projection rules relying on the available bridges are used for building a mate constraint [14, 6] which is submitted to the mate solver (think of $solve^{\mathcal{FD}}$ as the mate of $solve^{\mathcal{R}}$, and vice versa). Thus, projection enables each of the two solvers to take advantage of the constraints sent to the mate solver. In order to maximize the opportunities for projection, the goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules. Obviously, independent computing of solvers remains possible.

The goal solving rules in [14] describe the process of solver cooperation by means of the creation of new bridge constraints stored in M with the aim of enabling projections of mate constraints via bridges. Solver cooperation can be enabled only for bridges and also for both bridges and projections, which allows

to analyze the trade-off between communication flow and performance gain, so that the user can decide the best option for a given program.

4.3 Example

We show a problem (taken from [16]) which requires the cooperation of an $\mathcal{FD}$ solver and a continuous domain solver. In this example there exists an electric circuit with some connected resistors (modelled with real variables) and there is a set of capacitors (modelled with $\mathcal{FD}$ variables). The goal consists of knowing which capacitor has to be used so that its voltage reaches the 99% of the final voltage given a time range. The $\mathcal{TOY}$ program formulating the problem is shown below.

```
include "cflpfd.toy"

ecircuit :: real -> int
ecircuit C = KI <==
  R1 == 10000,
  10000 <= R2, R2 <= 40000,
  R == R1*R2/(R1+R2),
  50000.0 <= R, R <= 80000.0,
  T == -(ln 0.01)*R*K/10000000.0 + ET,
  0.5 <= T, T <= 1.0, -C <= ET, ET <= C,
  belongs KI [10,25,50,100,200,500],
  KI #== K,
  labeling [] [KI]
```

In this program, some relational constraint operators have been used (`==` for strict equality, and `<=` for "less or equal than"). Further, a finite domain constraint `belongs` is used, which prunes the domain of `KI` to take values in the given list of capacitor values. An $\mathcal{FD}$ enumeration procedure is applied with `labeling`, which selects the predefined enumeration strategy over the single variable `KI`. Finally, a bridge is used to connect the $\mathcal{FD}$ variable `KI` and the real variable `K` (that represents the continuous value of the capacitor). Note that, due to the imprecision of the real solver, a coupled variable `ET` is added. Real variables `C` and `T` represent, respectively, an input tolerance parameter and the time.

The following goal computes which capacitor has to be used if we consider the time interval [0.5,1] (measured in seconds).

```
Toy(R+FD)> ecircuit 0.001 == K
      { K -> 25 }
      Elapsed time: 16 ms.
sol.1, more solutions (y/n/d/a) [y]?
      no
      Elapsed time: 0 ms.
```

4.4 Bridges

Our Prolog implementation of bridges codes the application of the transformation rules for $solve^{\mathcal{M}}$ in Tables 3 and 4 of [6]. Each defined function in $\mathcal{TOY}$ is implemented as a Prolog predicate with the following arguments: its function arguments (as many as its arity), the function result, and two arguments representing the input constraint store and the output (i.e., modified) constraint store. The code excerpt below shows the basic implementation of the constraint `bridge` (i.e., `#==`):

```
(1) #==(L, R, Out, Cin, Cout):-
(2)    hnf(L, HL, Cin, Cout1),
(3)    hnf(R, HR, Cout1, Cout2),
(4)    tolerance(Epsilon),
(5)    ( (Out=true,
(6)          Cout3 = ['#=='(HL,HR)|Cout2],
(7)          freeze(HL, {HL - Epsilon =< HR, HR =< HL + Epsilon} ),
(8)          freeze(HR, (HL is integer(round(HR)))));
(9)      (Out=false,
(10)         Cout3 = ['#/=='(HL,HR)|Cout2],
(11)         freeze(HL, (F is float(HL), {HR =\= F})),
(12)         freeze(HR, (0.0 is float_fractional_part(HR) ->
(13)                  (I is integer(HR), HL #\= I); true)))),
(14)   cleanBridgeStore(Cout3,Cout).
```

As the $\mathcal{TOY}$ constraint *bridge* has arity 2, its Prolog implementation has two first arguments: `L` and `R` for the left (integer) and right (real) arguments of `#==`, respectively. `Out` is the argument for the result of its evaluation. `Cin` and `Cout` are the arguments for the incoming and outcoming constraint store. This store implements the stores H and M, including constraints from both domains $\mathcal{H}$ and $\mathcal{M}$, i.e., disequality constraints for constructed terms, and bridges and antibridges, resp. Notice that there is no need of explicit accounting for equality constraints on $\mathcal{H}$ since they are handled by Prolog unification.

Lines `(2)` and `(3)` flattens both `L` and `R` by calculating their head normal forms (hnfs), which always delivers either a variable or a number, therefore ensuring that no suspensions will occur from line `(4)` on. So, this implements the demandness of these arguments: They are required to be a variable or a number for a bridge constraint relating them to be posted. In addition, note that a hnf computation may develop new $\mathcal{H}$ disequality constraints during narrowing that have to be added to its constraint store.

Note that a bridge constraint `X #== Y` accepts reification. This means that if the value for `Out` is `true`, then the constraint `X #== Y` is posted to the store M (line `(6)`), whereas if the value is `false`, then the complementary constraint (the antibridge `X #/== Y`) is otherwise posted (line `(10)`). Also, note that this constraint can be used to impose an integral constraint over its right argument.

Implementing both `X #== Y` and `X #/== Y` is accomplished by using the concurrent predicate `freeze` available in SICStus Prolog. This predicate suspends the evaluation of its second argument until the first one becomes ground.

For the first case (`#==`), we need to reflect in this constraint the equality of its two arguments (variables or constants), which are of different type, i.e., real and integer, so that type casting is needed. `HR` is assigned to the float version of `HL` (line `(7)`) and `HL` is is assigned to the integer version of `HR` (line `(8)`). But, due to the imprecision nature of real solvers, occasionally, `HR` must take an approximation to an integer value. So, in order to avoid failures due to requiring exact integer values, it is necessary to introduce a tolerance value via the user-defined parameter `Epsilon` (line `(4)`), which is zero by default. Casting from floats to integers is performed by the Prolog operators `round` and `integer` (line `(8)`).

For the second case (`#/==`), we have to state in $solve^{\mathcal{M}}$ that both arguments are not equal, which cannot be directly handled, as before. So, whenever an argument becomes (or is) ground in a domain $\mathcal{FD}$ or $\mathcal{R}$, then a disequality constraint between the casted ground variable and its mate variable can be posted to the underlying solver (lines `11-13`).

Finally, the store is cleaned of ground bridges (i.e., variable-free) in line `(14)`. As well, variables occurring at the same end of two bridges are unified whenever the variables occurring at the other end become unified.

4.5 Projection: $\mathcal{FD}$ to $\mathcal{R}$

Projecting a constraint in $\mathcal{FD}$ to $\mathcal{R}$ is possible if the user has enabled projection and the constraint is allowed to be projected (see Table 1 in [14] or Table 3 in [6]). The projection amounts to, first, create bridges for the rest of variables in the $\mathcal{FD}$ constraint that are not involved in bridges, therefore creating new $\mathcal{R}$ variables with integral values which may be further related in other $\mathcal{R}$ constraints, and, second, send a mate constraint from $\mathcal{FD}$ to $\mathcal{R}$. The code excerpt below shows its basic implementation for the concrete constraint `#>` (i.e., greater than):

```
(1) #>(L, R, Out, Cin, Cout):-
(2)   hnf(L, HL, Cin, Cout1),
(3)   hnf(R, HR, Cout1, Cout2),
(4)   ((Out=true,  HL #>  HR);
(5)    (Out=false, HL #=< HR)),
(6)   toSolverFD(HL,Cout2,Cout3),
(7)   toSolverFD(HR,Cout3,Cout4),
(8)   (proj_active -> (
(9)      searchVarsR(HL,Cout4,Cout5,HLR),
(10)     searchVarsR(HR,Cout5,Cout,HRR),
(11)     ((Out==true, { HLR > HRR });
(12)      (Out==false, { HLR =< HRR }))));
(13)   Cout=Cout4).
```

This code implements the application of the rules for the cooperation (see Table 4 of [6]) among solvers. It follows the same prototype (line `(1)`) as `#==`, since it is a binary function. Its two input arguments (`L` and `R`) are demanded to be in hnf (lines `(2-3)`), and a primitive constraint is posted to the underlying

$\mathcal{FD}$ solver, depending on the Boolean result of the function (lines `(4)-(5)`). Moreover, these variables can be involved in (Herbrand) disequality constraints because they were not identified as $\mathcal{FD}$ variables already. If so, $\mathcal{FD}$ disequality constraints are also posted to $solve^{\mathcal{FD}}$ (lines `(6-7)`). This scenario appears because the type of a variable is not always known since types are checked and inferred at compile-time, but this information is not present at run-time.

If projection is active (indicated by the dynamic predicate `proj_active` in line `(8)`), then bridges relating the arguments of `#>` are looked for in the mediatorial store in order to find mate variables in $\mathcal{R}$ (lines `(9-10)`). This search, if unsuccessful, will otherwise create bridges relating new mate variables in $\mathcal{R}$. Finally, a mate constraint is sent to the underlying $\mathcal{R}$ solver (lines `(11-12)`).

4.6 Projection: $\mathcal{R}$ to $\mathcal{FD}$

Projecting a constraint from $\mathcal{R}$ to $\mathcal{FD}$ is possible in the same conditions stated in the previous section. The projection amounts to send mate constraints as before, but bridges for the rest of variables in an $\mathcal{R}$ constraint are not created since their integral nature is not for sure. The code excerpt below shows its basic implementation (without considering obvious optimizations) for a concrete constraint `>` (i.e., greater than):

```
(1) >(L, R, Out, Cin, Cout):-
(2)  hnf(L, HL, Cin, Cout1),
(3)  hnf(R,HR, Cout1, Cout2),
(4)  (Out = true, {HR > HL} ;
(5)   Out = false, {HL =< HR}),
(6)  toSolver(HL, Cout2, Cout3),
(7)  toSolver(HR, Cout3, Cout4),
(8)  toSolver(Out, Cout4, Cout),
(9)  (proj_active ->
(10)    (searchVarsFD(HL, Cout, BL, FDHL),
(11)     searchVarsFD(HR, Cout, BR, FDHR),
(12)     ((BL == true,  BR == true,  Out == true,  FDHL #> FDHR);
(13)      (BL == true,  BR == true,  Out == false, FDHL #=< FDHR);
(14)      (BL == true,  BR == false, Out == true,  FDHL #> FDHR);
(15)      (BL == true,  BR == false, Out == false, FDHL #=< FDHR);
(16)      (BL == false, BR == true,  Out == true,  FDHL #>= FDHR);
(17)      (BL == false, BR == true,  Out == false, FDHL #< FDHR);
(18)      true);
(19) true).
```

After analogous steps to the previous subsection, lines `(6)-(8)` deal with the explicit interaction between $solve^{\mathcal{H}}$ and $solve^{\mathcal{R}}$ [17], checking whether the disequality affects a real variable; if so, the constraint is sent to the underlying solver for reals and removed from the Herbrand store.

Next, if projection is active, a similar procedure to the one performed for the projection in the other direction follows. However, notice that there are more possibilities for sending a mate constraint to $solve^{\mathcal{FD}}$ (see Table 4 in [6]),

depending on values of BL, BR, and Out. For BL and BR, a true value means that a bridge relating this variable has been found; a false value means that HL (resp. HR) is a real value with non-zero fractional part. Following the lines (10)-(11), FDHL (resp. FDHR) is the greatest integral value less or equal to HL (resp. HR).

Lines (12)-(13) correspond to true values for BL and BR therefore the mate constraint #> is sent to $solve^{\mathcal{FD}}$, or its counterpart #=<, depending on the Boolean result of the function.

Lines (14)-(17) determine the correct mate constraint which is sent to $solve^{\mathcal{FD}}$, which is selected in terms of the values for BL, BR, and Out, as specified in [6]. For example, line (14) is selected for solving the right argument of the conjunctive goal X #== RX, RX > 4.3. Here, BL is true as the real variable RX has a mate finite variable X, (FDHL is X), BR is false because 4.3 has a non-zero fractional part (so, FDHR is 4). Finally, the mate constraint X #> 4 is posted to $solve^{\mathcal{FD}}$.

4.7 Handling of Numerical Types

$\mathcal{TOY}$ is a typed programming language, based essentially on the Hindley-Milner-Damas polymorphic type system [18]. Programs are tested for well-typedness at compile time. In particular, each occurrence of an expression in a $\mathcal{TOY}$ program has a type that can be determined at compile time. Syntactically, types are built from *type variables* $tvar(\tau)$ and *type constructors* TC. Any identifier starting with an uppercase letter can be used as a type variable, while identifiers for type constructors must start with a lowercase letter. Type constructors are introduced in *datatype declarations*, along with data constructors. Primitive types (such as bool, int, and real) can be viewed as type constructors of arity 0.

Function symbols are required to come along with a so-called *principal type declaration*, which indicates its most general type. For example, the types of the function "greater than" are >::real -> real -> bool and #>::int -> int -> bool, distinguishing the $\mathcal{FD}$ version from the real number one by the prefix symbol #; the exception is the equality and disequality constraints (== and /== respectively) that are overload in order to work in both domains.

Type of variables is not always known at run-time because type inference information is not kept. Thus, a disequality constraint between variables is assumed to range over $\mathcal{H}$, so that it is sent to the Herbrand store. During the constraint solving process, $\mathcal{FD}$ and real constraints are continuously involved in a projection process that gives rise to the update of different $\mathcal{FD}$ and real variables; as a consequence, the disequality constraints stored may be affected by the updates on any finite domain or real variable; if so, each of these disequality constraints is sent to the underlying solver for $\mathcal{FD}$ or real domain in order to look for inconsistences.

Equality constraints are treated differently since these are handled by unification. Narrowing process reduces both arguments to hnf, but $\mathcal{TOY}$ uses a sophisticated process that analyzes the structure of both arguments in order to cut the search space.

In our system, some problems arose with types in different scenarios. For example, we can solve the goals `X+2>4` and `X+2/=4`, but the goal `X+2==4` throws an exception because it tries to unify a real value (i.e., the result of the narrowing of the expression `X+2`) with the integer value `4`. This can be avoided in $\mathcal{TOY}$ by identifying correctly the nature of the result; in the example, the exception is not thrown if we type `X+2==4.0` (note that the left argument of the equality provides always a real value as function `+` is defined as `+::real -> real -> real` and thus the value 2 is interpreted as a real value).

4.8 Related Work

In general, solver cooperation has been widely analyzed in the literature and there are a number of declarative constraint systems that provide support for the interaction among solvers. For example: CLP(BNR) [19], Prolog III [20] and Prolog IV [21] allow solver cooperation, mainly limited to Booleans, reals and naturals. Also, the language NCL [22] provides an integrated constraint framework that strongly combines Boolean logic, integer constraints, and set reasoning. The integration of new constraint domains such as the reals is described as future work in [22]. In general, all those systems provide a limited form of cooperation that is very specific to the predefined computation domains existing in the system. Solver cooperation as integrated in $\mathcal{TOY}$ is quite different from all those systems as its implementation follows the theoretical principles recently described in [6]. Particularly, solver cooperation in $\mathcal{TOY}$ follows an *interoperative approach*, which means that the system has the ability to communicate and use independently-written software components, thus allowing independent systems to cooperate. In the literature, one can find different proposals catalogued in this approach. For instance, [23] proposes a C++ constraint solving library called aLiX for communicating different solvers, possibly written in different languages. One of the main shortcomings of the current aLiX version is that a component for solving continuous constraints is not integrated into the system yet (this is claimed to be one of their main priorities for future development work).

Also, [24] describes a client/server architecture to enable communication among the component solvers. This consists of both managers of the system and the solvers that must be defined on the same computational domain (e.g., real numbers) but with different classes of admissible constraints (e.g., linear and non-linear constraints). The CLP system *CoSAc* is an implementation of their system. This system is very different from our proposal as the exchange of information is managed by means of pipes and the exchanged data is a character string. Also, in his thesis [25], Monfroy constructed the system **BALI** (Binding Architecture for Solver Integration) that facilitates the integration of heterogeneous solvers, as well as the specification of solver cooperation via a number of cooperations primitives. There are may differences with our implementation but one of the most significant is that Monfroy's approach assumes that all the solvers work over a common store, while our present proposal requires communication among different stores.

Perhaps, regarding solver cooperation, the most similar system to $\mathcal{TOY}$ is the system `Meta-S` [26], a meta-solver framework that implements the ideas proposed in [27] for the dynamic integration of external stand-alone solvers to enable the collaborative processing of constraints. The similarities between $\mathcal{TOY}$ and `Meta-S` are because solver cooperation in `Meta-S` also relies on two main constructs, namely constraint propagation (that enables to submit a constraint belonging to some domain $\mathcal{D}$ to its constraint store, say $S_{\mathcal{D}}$) and projection of constraint stores (that consults the contents of a given store $S_{\mathcal{D}}$ and deduces constraints for another domain). Our projection differs from `Meta-S` projection in the creation and use of bridges; `Meta-S` propagation corresponds to our goal solving rules for placing constraints in stores and invoking constraint solvers. An important difference is the lack of bridges in `Meta-S` approach that corresponds to the lack of mediatorial domains within the combined domains that can be constructed in this system. From the implementation point of view, there are additional structural differences between $\mathcal{TOY}$ and `Meta-S`. So, `Meta-S` does not provide facilities for constraint optimization (as $\mathcal{TOY}$). Also, `Meta-S` is implemented in Common Lisp whereas $\mathcal{TOY}$ is implemented in Prolog.

5 Performance

In this section, we briefly show empirically that the projection mechanism of $\mathcal{TOY}$ helps to accelerate the cooperative constraint solving. To do so, we have considered a number of benchmarks[4]: a *non-linear crypto-arithmetic (nl-csp)* problem (9 $\mathcal{FD}$ variables with non-linear equations), two problems for solving systems of 10 (*eq10*) and 20 (*eq20*) linear equations with 7 $\mathcal{FD}$ variables, an *electrical circuit* problem, a *knapsack* optimization problem, and a set of cryptoarithmethic problems i.e., *send+more=money (smm)* problem (8 $\mathcal{FD}$ variables, 1 linear equation, 2 disequations, and 1 *all different* constraint), the *Wrong+Wrong=Wright* (wwr) problem (8 $\mathcal{FD}$ variables, 1 linear equation, 1 *all different* constraint), the *alpha* problem (26 $\mathcal{FD}$ variables, 20 linear equations, and 1 *all different* constraint), and the *donald* problem (10 $\mathcal{FD}$ variables, 1 linear equations, and 1 *all different* constraint). All the benchmarks were coded to require $\mathcal{FD}$ constraint solving as well as solving of (non-)linear equations in $solve^{\mathcal{R}}$.

All the benchmarks were executed on the same Linux machine, operating system Suse Linux 9.3, with an Intel(R) Pentium(R) M processor running at 1.70GHz and with a RAM memory of 1 GB. For the sake of brevity, in Table 1 we only provide the results for first solution search. The first column displays the configuration employed: (1) $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$, which corresponds to $\mathcal{TOY}$ with both numerical solvers activated, and (2) $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$-proj which corresponds to $\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$ with the mechanism for constraint projections activated. In addition, two labeling strategies were considered: *naïve*, in which variables are labelled in a prefix order, and *first fail (ff)*, in which the variable with the smallest domain is chosen first for enumerating. The label ($\mathcal{FD} \sim \mathcal{R}$)

[4] All the benchmarks are available in `http://www.lcc.uma.es/~afdez/cflpfdr`.

means that labelling of $\mathcal{FD}$ variables and global constraints are executed in $solve^{\mathcal{FD}}$ whereas (non-)linear-equations are sent to $solve^{\mathcal{R}}$. The numbers for the different versions of $\mathcal{TOY}$ represent the average of ten runs. Note that, in general, activating the projection mechanism provides a significant performance improvement. Further experiments with more benchmarks have also been executed leading to the same conclusion.

Configuration	knapsack	donald	smm	nl-csp	wwr	eq10	eq20	alpha	circuit
$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$									
naïve $\mathcal{FD} \sim R$	16	304970	22528	411	411	266	402	314	14
ff $\mathcal{FD} \sim R$	15	288700	22627	383	420	271	408	272	**13**
$\mathcal{TOY}(\mathcal{FD}+\mathcal{R})$-proj									
naïve $\mathcal{FD} \sim R$	11	8305	41	44	54	290	433	291	14
ff $\mathcal{FD} \sim R$	16	601	40	87	58	269	397	283	20
Speed-Up									
naïve $\mathcal{FD} \sim R$	1.45	36.72	549.43	9.34	7.61	0.91	0.92	1.07	1
ff $\mathcal{FD} \sim R$	0.93	480.36	565.67	4.4	7.24	1	1.02	0.96	0.65

Table 1. Running time (milliseconds) for first solution search

6 Conclusions and Future Work

In this paper, we have dealt with implementation issues of the constraint functional logic programming system $\mathcal{TOY}$ unreported up to now. Among these implementation issues, we have described the data-flow program compilation process, available libraries, and the integration of constraint solving technology in the system. With special emphasis, we have explained how solver cooperation has been recently incorporated in $\mathcal{TOY}$. This is a very important issue as the interaction among solvers makes it easier to express compound problems, and good communication can help the efficiency of the systems [28].

More specifically, we have described the internal communication among $\mathcal{H}$, $\mathcal{R}$ and $\mathcal{FD}$ via bridges and projections. We have sketched their implementation, and shown that bridges manage the communication between two variables that belong to different computation domains, whereas propagation generates, from a primitive constraint defined on one source computation domain, new (semantically-equivalent) constraints that are propagated to another computation domain which demands cooperation with the source domain. This solver cooperation can lead to drastic reductions in the search space of the problem, and can be translated into a reduction of the solving time as it was shown in [6].

Acknowledgements

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 TIN2004-7943-C04-01 and TIN2005-08818-C04-01.

References

1. Arenas, P., Fernández, A., Gil, A., López-Fraguas, F., Rodríguez-Artalejo, M., Sáenz-Pérez, F.: $\mathcal{TOY}$. A Multiparadigm Declarative Language. Version 2.3.0 (July 2007) R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.
2. González-Moreno, J., Hortalá-González, M., López-Fraguas, F., Rodríguez-Artalejo, M.: An Approach to Declarative Programming Based on a Rewriting Logic. The Journal of Logic Programming **40**(1) (July 1999) 47–87
3. Fernández, A.J., Hortalá-González, T., Sáenz-Pérez, F., del Vado-Vírseda, R.: Constraint Functional Logic Programming over Finite Domains. Theory and Practice of Logic Programming (2007) In Press.
4. Hortalá-González, T., López-Fraguas, F., Sánchez-Hernández, J., Ullán-Hernández, E.: Declarative Programming with Real Constraints (1997) Technical Report SIP 5997, Univ. Complutense Madrid, 1997.
5. Arenas, P., Gil, A., López-Fraguas, F.: Combining Lazy Narrowing with Disequality Constraints. In: PLILP'94, Springer LNCS 844 (1994) 385–399
6. Estévez-Martín, S., Fernández, A., Hortalá-González, T., Rodríguez-Artalejo, M., Sáenz-Pérez, F., del Vado-Vírseda, R.: A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming. ENTCS **188** (2007) 37–51
7. SICStus Prolog: (2006) `http://www.sics.se/isl/sicstus`.
8. Antoy, S., Hanus, M.: Compiling Multi-Paradigm Declarative Programs into Prolog. In: Frontiers of Combining Systems. (2000) 171–185
9. Loogen, R., López-Fraguas, F., Rodríguez-Artalejo, M.: A Demand Driven Computation Strategy for Lazy Narrowing. In: PLILP. (1993) 184–200
10. Antoy, S.: Definitional Trees. In: 3rd International Conference on Algebraic and Logic Programming (ALP'92). Number 632 in LNCS, Volterra, Italy, Springer-Verlag (1992) 143–157
11. Caballero, R., López-Fraguas, F.J.: Dynamic-Cut with Definitional Trees. In: FLOPS '02: Proceedings of the 6th International Symposium on Functional and Logic Programming, London, UK, Springer-Verlag (2002) 245–258
12. Caballero, R.: A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In: WCFLP '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, ACM Press (2005) 8–13
13. López-Fraguas, F., Rodríguez-Artalejo, M., del Vado-Vírseda, R.: A Lazy Narrowing Calculus for Declarative Constraint Programming. In: Proc. of PPDP'04, ACM Press (August 2004) 43–54
14. Estévez-Martín, S., Fernández, A.J., Hortalá-González, M.T., Rodríguez-Artalejo, M., del Vado Vírseda, R.: A fully sound goal solving calculus for the cooperation of solvers in the cflp scheme. ENTCS **177** (2007) 235–252

15. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A Multiparadigm Declarative System. In Narendran, P., Rusinowitch, M., eds.: RTA. Number 1631 in LNCS, Springer (1999) 244–247
16. Hofstedt, P.: Cooperation and Coordination of Constraint Solvers. Phd thesis, Technischen Universität Dresden, Fakultät Informatik (2001)
17. Sánchez-Hernández, J.: $\mathcal{TOY}$: Un Lenguaje Lógico funcional con restricciones (1998) Available (in Spanish) at `http://babel.dacya.ucm.es/jaime/publications.html`.
18. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL'82, ACM Press (1982) 207–212
19. Benhamou, F., Older, W.: Applying interval arithmetic to real, integer and Boolean constraints. The Journal of Logic Programming **32**(1) (July 1997) 1–24
20. Colmerauer, A.: An introduction to PROLOG III. Communications of the ACM (CACM) **33**(7) (July 1990) 69–90
21. N'Dong, S.: Prolog IV ou la programmation par contraintes selon PrologIA. In: Sixièmes Journées Francophones de Programmation Logique et Programmation par Contraintes (JFPLC'97), Orléans, France, Edition HERMES (1997) 235–238
22. Zhou, J.: Introduction to the constraint language NCL. The Journal of Logic Programming **45**(1-3) (2000) 71–103
23. Goualard, F.: Component programming and interoperatibility in constraint solver design. In K.Apt, Barták, R., Monfroy, E., Rossi, F., eds.: ERCIM Workshop on Constraints, Prague, Czech Republic, Charles University/Faculty of Mathematics and Physics (2001)
24. Monfroy, E., Rusinowitch, M., Schott, R.: Implementing non-linear constraints with cooperative solvers. Research Report 2747, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (December 1995)
25. Monfroy, E.: Solver collaboration for constraint logic programming. PhD thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (November 1996)
26. Frank, S., Hofstedt, P., Reckman, D.: Meta-s - combining solver cooperation and programming languages. In Wolf, A., Frühwirth, T.W., Meister, M., eds.: Proc. W(C)LP 2005, Ulmer Informatik-Berichte 2005-01 (2005) 159–162
27. Hofstedt, P., Pepper, P.: Integration of Declarative and Constraint Programming. Theory and Practice of Logic Programming (2006) In Press.
28. Granvilliers, L., Monfroy, E., Benhamou, F.: Cooperative solvers in constraint programming: a short introduction. ALP Newsletter **14**(2) (May 2001)