DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA

LCC ITI 06-8

INFORME TÉCNICO

INVESTIGACIÓN

Implementing $\mathcal{TOY}$, a Constraint Functional Logic Programming with Solver Cooperation

Sonia Estévez-Martín, F. Sáenz-Pérez

Sistemas Informáticos y Programación,

Universidad Complutense de Madrid

Madrid, Spain

e-mails: s.estevez@fdi.ucm.es, s.estevez@fdi.ucm.es

Antonio J. Fernández

Lenguajes y Ciencias de la Computación,

Universidad de Málaga,

Málaga, Spain

e-mail: afdez@lcc.uma.es

November 8, 2006

Implementing $\mathcal{TOY}$, a Constraint Functional Logic Programming with Solver Cooperation

S. Estévez-Martín[a,1] A. J. Fernández[b,2] F. Sáenz-Pérez[a,1]

[a] *Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid*
{s.estevez,fernan}@sip.ucm.es

[b] *Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga*
afdez@lcc.uma.es

Abstract

This paper describes implementation issues of the constraint functional logic system $\mathcal{TOY}$, which has recently incorporated new features such as solver cooperation. The formal framework introduced in [14] is the basis for this implementation, allowing the cooperation of three constraint solvers over different domains: the Herbrand domain, the finite domain, and the real numbers domain. The system offers both binding and propagation as mechanisms for the cooperation. The paper describes the implementation of the system and, in particular, regarding solver cooperation, presents a new function that serves as a bridge among the solvers involved in the cooperation. The new system ($\mathcal{TOY}$ version 2.3.0) is built upon the last software release ($\mathcal{TOY}$ version 2.2.3), inheriting all of the previous features. The contents of this paper covers the aspects of the implementation which have not been reported elsewhere. We describe its software architecture, data-flow of compilation, loading of libraries, and the implementation of binding and propagation.

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

1 Introduction

The design, implementation, and optimization of declarative (particularly logic) constraint programming systems can be considered one of the major issues treated in recent years in the constraint programming and logic programming areas. Proof of it is the annual celebration of the (already stable) colloquium on implementation of constraint and logic programming systems (i.e., CICLOPS) that is usually held as a satellite workshop of the international conference on logic programming.

One of the main reasons for this interest in the implementation of declarative constraint systems is the continuous progress that the computing technology is experimenting nowadays. This progress is translated into an improvement of physical

[1] Author partially supported by projects TIN2005-09207-C03-03 and S-0505/TIC0407.
[2] Author partially supported by projects TIN2004-7943-C04-01 and TIN2005-08818-C04-01.

factors that affect directly to the performance of a programming system, e.g., faster processor or memories with higher capacities, among other factors. Additionally, declarative constraint languages (e.g., CHR [19], Prolog [31,9,13], Curry [24], Oz (Mozart) [34], HAL [11] or $\mathcal{TOY}$ [28], among others) are considered as high level programming tools that ease the task of programming (e.g., formulation of the problem or program analysis) and provide a reasonable balance between compile-time effort and run-time problem solving. The addition of constraint solving technology to declarative programming systems has caused that these systems are now being considered good options for programmimg complex and real problems. 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 paper, we consider specifically the constraint functional logic programming (CFLP) language $\mathcal{TOY}$ [4,20]. 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 [15]), support for managing arithmetic linear and non-linear constraints defined on the real domain $\mathcal{R}$ [26], and provision of strict equality and disequality constraints [5] defined in the Herbrand domain $\mathcal{H}$. The domain-specific constraints are solved in the associated domain-specific solvers (i.e., Solver$^{\mathcal{FD}}$, Solver$^{\mathcal{R}}$, Solver$^{\mathcal{H}}$, respectively) that are connected to the system via an adequate interface.

Recently, with the aim of extending the applicability of the system, $\mathcal{TOY}$ has incorporated new features such as solver cooperation. The implementation of this feature in $\mathcal{TOY}$ is based on the theoretical framework described in [14]. Basically, this cooperation allows the communication between Solver$^{\mathcal{FD}}$ and Solver$^{\mathcal{R}}$ by means of special communication constraints called *bridges*. A bridge `u #== v` constrains `u::int` and `v::real` to take the same integer value. Bridges are kept in a special store and they are used for two purposes, namely *binding* and *propagation*. Binding simply instantiates a variable occurring at one end of a bridge whenever the other end becomes a numeric value. Propagation is a more complex operation which takes place whenever a pure constraint is submitted to Solver$^{\mathcal{FD}}$ or Solver$^{\mathcal{R}}$. At that moment, propagation rules relying on the available bridges are used for building a mate constraint which is submitted to the mate solver (think of Solver$^{\mathcal{FD}}$ as the mate of Solver$^{\mathcal{R}}$, and viceversa). Propagation enables each of the two solvers to take advantage of the computations performed by the other. In order to maximize the opportunities for propagation, the goal solving procedure has been enhanced with operations to create bridges whenever possible, according to certain rules. Obviously, independent computing of solvers remains possible.

Goal solving takes care of evaluating expressions to functions by means of lazy narrowing [12,27], and decomposing constraints by introducing new local variables.

Eventually, primitive $\mathcal{FD}$ and $\mathcal{R}$ constraints arise, which must be submitted to the respective solvers. $\mathcal{TOY}$ has been implemented on top of SICStus Prolog [33], using the $\mathcal{FD}$ and $\mathcal{R}$ solvers provided by SICStus along with Prolog glue code for interfacing them with Solver$^{\mathcal{FD}}$ and Solver$^{\mathcal{R}}$, respectively, code for implementing Solver$^{\mathcal{H}}$, and code for implementing lazy narrowing dealing with constraints.

This paper focuses specifically on implementation issues, not reported so far, of the $\mathcal{TOY}$ system. Among these issues, the paper provides implementation details about the architecture of the system, its module system, program compilation dataflow, the way in which libraries are loaded in the system and, more particularly, how the solver cooperation described in [14] 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.

2 Software Architecture

$\mathcal{TOY}$ [14] has four constraint domains available (see Figure 1): $\mathcal{H}$ 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 $\mathcal{M}$ for bridge constraints between numerical solvers, allowing its cooperation by means of binding and propagation.

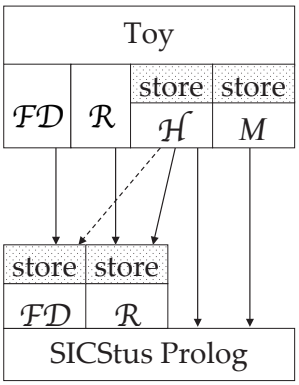

Fig. 1. Software Architecture of $\mathcal{TOY}$

Each constraint domain has an attached constraint store (H, R, FD, and M for $\mathcal{H}$, $\mathcal{R}$, $\mathcal{FD}$, and $\mathcal{M}$, respectively) and solver (Solver$^{\mathcal{H}}$, Solver$^{\mathcal{R}}$, Solver$^{\mathcal{FD}}$, and Solver$^{\mathcal{M}}$). We take advantage of the SICStus Prolog [33] constraint stores for storing $\mathcal{R}$ and $\mathcal{FD}$ constraints. For the sake of rapid prototyping, we have implemented the stores H and M as a single list, although they should be separated for better performance. In addition, we always add constraints to the communication store (irrespective of groundness) and never drop them (again to be enhanced in a forthcoming release).

Primitive constraints for $\mathcal{R}$ and $\mathcal{FD}$ are posted to Solver$^{\mathcal{R}}$ and Solver$^{\mathcal{FD}}$, respectively, which are already implemented in SICStus Prolog libraries. The impedance mismatch problem between the host language and these solvers is tackled by glue code (cfr. Section 5). Bridge constraints for $\mathcal{M}$ are implemented as introduced

in [14] and detailed in Section 5. Equality and disequality constraints for $\mathcal{H}$ are implemented as already reported [5,1].

Disequality constraints may affect a variable whose type is unknown [28]. These constraints are assumed to involve constructed terms and, therefore handled by Solver$^{\mathcal{H}}$. But, along computation, these variables may be instantiated to a number, so that the corresponding disequality constraints are moved to Solver$^{\mathcal{R}}$. In fact, this should also be done with Solver$^{\mathcal{FD}}$ for integers, but we have not dealt with its implementation up to now. Since a disequality constraint (`#/=`) in Solver$^{\mathcal{FD}}$ is syntactically different from a disequality contraint in Solver$^{\mathcal{R}}$ and Solver$^{\mathcal{H}}$ (`/=`), programmers are warned about this issue.

2.1 Modules

The $\mathcal{TOY}$ system has several usage modes (with respect to the loaded libraries and active constraint solvers) which either imply to have more functions available, or a different implementation of existing functions, or removing existing functions. We have thus taken advantage of the module system provided by SICStus Prolog, which allows to easily interchange, add or remove the implementation of predefined functions by simply loading a file, besides the advantages of modularization for software development. Each module is defined by a file which consists of the module declaration, import and export lists, and the definition of exports and hidden predicates. Next, we indicate some of the modules which are mainly implied in particular tasks, and enclosing between parenthesis the file which define the module.

- System initialization and operation:
  `toy` (`toy.pl`): The first module loaded and used for initialization purposes.
  `initToy` (`initToy.pl`): For interpreting commands and evaluate expressions from the command prompt.
  `process` (`process.pl`): For the compilation process.
  `dyn` (`dyn.pl`): Contains dynamic predicates which indicates the system state. For instance, `prop_active` for indicating whether propagation between numerical solvers is active, or `cflpfd_active` for indicating whether the finite domain library is loaded.
  `tools` (`tools.pl`): Utility predicates.
  `toycomm` (`toycomm.pl`): Contains common predicates to $\mathcal{TOY}$ programs (e.g., the predicate `hnf`).
- Compilation:
  `compil` (`compil.pl`): Lexical, syntactical, (part of) semantical analysis, and dependency analysis of functions.
  `codfun` (`codfun.pl`): Generates the Prolog code of a function using the definitional tree.
  `errorToy` (`errorToy.pl`): Error handling during compilation.
  `inferrer` (`inferrer.pl`): Type inference and checking.
  `outgenerated`: Contains defined and predefined type and function definitions, definitional trees [2], and flags for indicating deterministic functions. It is built up and loaded during program compilation (see Section 3).
- Built-ins and goal solving (see Section 4):

`plgenerated` (`basic.pl`): Intended as a $\mathcal{TOY}$ user module in which the result of compiled code is available for goal solving. It contains type definitions for predefined constructors and functions, including the code to implement the operational behaviour of predefined functions (for instance, the code for suspensions and partial applications). Its contents vary depending on whether a user program is compiled (see Section 3) or a library (file, graphics, reals, finite domain (see Subsection 4.3)) is loaded.

`primFunct` (`primFunct.pl`): Predefined functions (infix and type declarations) are defined in this module. The contents of this module depend on the libraries which have been loaded, so that they are built at run-time from different files.

`primitivCod` (`primitivCod.pl`, `primitivCodClpr.pl`): The operational behaviour of predefined functions is implemented in this module, which again varies with the library loaded. If the library for reals is loaded, the definition of arithmetical operators is different since they are handled in constraint expressions which are sent to Solver$^{\mathcal{R}}$.

`primitivCodIo` (`primitivCodIo.pl`): Primitives for the file library.

`primitivCodGra` (`primitivCodGra.pl`): Primitives for the graphics library.

3 Compiling Programs

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 for running compiled $\mathcal{TOY}$ programs, as done in other related systems [3]. The compilation follows a demand driven computation strategy for lazy narrowing [27]. In this section, instead of describing the compilation (which was already reported [27,1]), we rather describe the data-flow involved in this procedure.

Given the $\mathcal{TOY}$ program `program.toy`, its compilation follows the data-flow depicted in Figure 2. First, defined functions, types and constructors in `program.toy` are joined with predefined ones in `basic.toy`, giving the file `program.tmp.toy`.

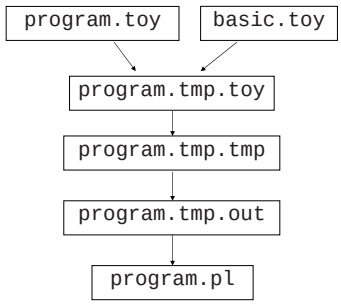

Fig. 2. Compilation Data-Flow

Next, a first compilation stage is performed: lexical, syntactical, and (part of) semantical analysis, giving the file `program.tmp.tmp`, which in particular contains the result of higher-order to first-order program translation [21]. A second compilation stage gives the file `program.tmp.out`, which defines the module `outgenerated` containing type declarations, functional dependencies, and definitional trees [2]. This file is consulted, in particular allowing the user to ask for definitional trees with the command `/tree`.

Finally, from the file `program.tmp.out`, a type inference is performed to infer non-declared types and check that user-declared types are correct. The final compilation stage is also performed in this step, generating the compiled (Prolog) code, giving the file `program.pl`, which defines the module `plgenerated`. This file, when next consulted, loads the files `toycomm.pl`, `primFunct.pl`, `primitivCod.pl` (or `primitivCodClpr.pl` if the constraint solver over reals is active) into `plgenerated` (see Section 4). It also loads `primitivCodIo.pl` and `primitivCodGra.pl` if the file and graphical libraries are loaded, respectively (see Section 4). `program.pl` also contains the Prolog code for the functions in `program.toy`. The last compilation stage departs from the definitional trees in `program.tmp.out`, performing a case analysis about the shape of these trees [27]. In addition, `program.pl` contains code for dynamic cut [8], type declarations for predefined functions and constructors, head normal form (hnf) calculations, partial applications and declarations of precedence and associativity for infix operators. The file `program.pl` is loaded for defining the (user) module `plgenerated` so that both defined (in `program.toy`) and predefined functions, types and constructors are available during goal solving. Note that if a program is compiled, its definitions in the module `plgenerated` will be lost if another program is compiled afterwards. Therefore, if definitions in several programs are needed, the `include` statement [4] has to be used.

4 Loading Libraries

Four libraries can be optionally loaded into $\mathcal{TOY}$: file, graphics, constraints over real numbers, and constraints over finite domains (integers). They add type, data constructor and function definitions to the basic system, therefore requiring more main memory. Loading more libraries means leaving less memory for goal solving, although this issue becomes less noticeable as main memory capacities grows along time. Note that loading the real numbers constraint library implies understanding arithmetical equations as logical relations, requiring no groundness on related variables, thus allowing a more declarative understanding of the program. However, this may cause a programmer familiarized with logic or functional programming without constraints to mistake tests for relations, which will be handled by the most costly constraint solving procedure. Next, we show the different ways libraries are loaded, which mainly depend on technical and rapid-prototyping reasons.

4.1 File and Graphics

Loading and unloading the file and graphics libraries are performed with the user commands `/io_file`, `/noio_file`, `/io_graphic` and `/noio_graphic`, respectively. Loading a library performs three main stages: first, files for defining modules are built; second, they are consulted in the Prolog underlying system; and, third, a void file is compiled to generate and load the module `plgenerated`. This allows both the compilation and execution of user programs using predefinitions in such libraries. In the following, we focus on loading the graphic library, an analogous process to the loading of the file library.

Loading the graphics library involves four modules: `primitivCod`, `primitivCodGra`, `primFunct`, and `plgenerated`, as shown in Figure 3. The contents of the files

defining modules depend on the loaded libraries, so that if, e.g., the library for real numbers is loaded, the file defining the module `primitivCod` is `primitivCodClpr.pl` instead of `primitivCod.pl`, as shown in the picture with the symbol $\oplus$, meaning that only one of the alternative files will be used for defining the module. On the other hand, there are other modules (`primFunct` and `plgenerated`) which are built by joining in one file the contents of, at least, two files (`basiccopia.pl` and `basicGra.pl`) for loading the graphics library. If the file library is already loaded, the joining will also include the file `basicIo.pl` (the optional inclusion of a file is shown in the figure by enclosing the file name in a dashed box).

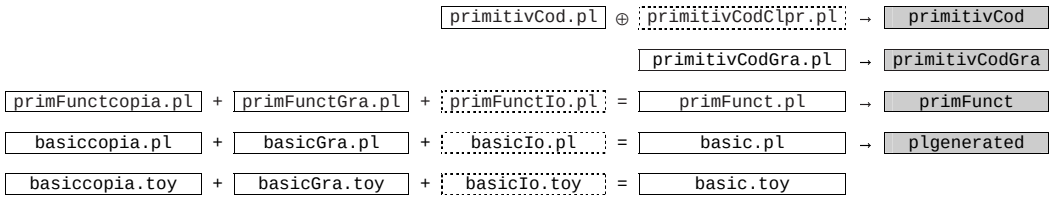

Fig. 3. Modules and Built-In Definitions for the Graphics Library

These modules make predefinitions available to the system at the command prompt, so that goals containing built-ins can be compiled and executed since predefinitions are visible from the built modules. In addition, the file `basic.toy` is built up from, at least, the files `basiccopia.toy` (containing predefinitions of basic types and functions) and `basicGra.toy` (containing predefinitions of types and functions for graphics). This allows to have available predefinitions in the graphics library for building up a temporary file during the compilation (cfr. Section 3).

Each module is then built up by consulting the corresponding files (cfr. Section 2). Then, the void file `nada.toy` is compiled, yielding the file `nada.pl`, in which the module `plgenerated` is defined. The contents of this file are similar to `basic.pl`.

Loading the file library is similar to loading the graphics library, with the module `primitivCodIo` instead of `primitivCodGra`, and the files `primitivCodIo.pl`, `primFunctIo.pl`, `basicIo.pl`, and `basicIo.toy` instead of `primitivCodGra.pl`, `primFunctGra.pl`, `basicGra.pl`, and `basicGra.toy`, respectively.

Unloading these libraries amounts to reload a given loading configuration; that is, given the loaded libraries indicated in the system state (module `dyn`, cfr. Section 2), a loading process is performed omitting the just unloaded library. This applies to the libraries for files, graphics, and constraints for real numbers, but it is different for the finite domain constraint library as explained in Subsection 4.3.

4.2 Constraints over Reals

Loading and unloading the real numbers library are performed with the user commands at the system prompt `/cflpr` and `/nocflpr`, respectively. The loading command performs the same stages as for loading files or graphics libraries.

Since the operational behaviour of arithmetical operators changes, its implementation is redefined in the module `primitivCod` (see Figure 4), taking `primitivCodClpr.pl` as its defining file. In addition, this file defines four new optimization functions: `minimize`, `maximize`, `bb_minimize`, and `bb_maximize`, the first two are intended for linear programming optimization, and the second two for mixed integer

linear programming optimization (using a branch and bound algorithm). Therefore, there is no a specific module for this library because the module `primitivCod` is used for this purpose. However, modules `primFunct` and `plgenerated` are built up in a similar way as indicated in the previous section. Also, the file `basic.toy` is to be built up in a similar way with, in particular, predefinitions of reals.

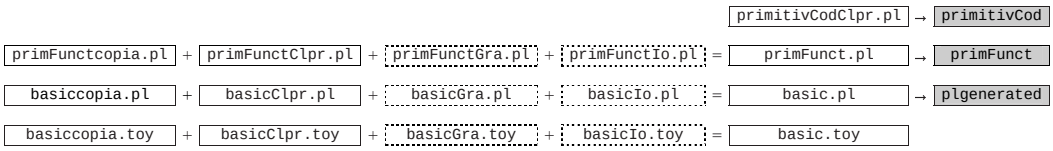

Fig. 4. Modules and Built-In Definitions for the Real Numbers Constraint Library

The file `primitivCodClpr` also includes the loading of the SICStus Prolog constraint logic programming over reals library, which makes available the underlying Prolog system solver to compiled $\mathcal{TOY}$ goals and programs.

4.3 Constraints over Finite Domains

Loading and unloading the library for constraints over finite domains are performed with the user commands at the system prompt `/cflpfd` and `/nocflpfd`, respectively. In contrast to the loading and unloading processes just described in previous subsections, we have followed a different point of view in order to promote rapid prototyping.

The objective is to have available both the $\mathcal{TOY}$ library for finite domain constraints and the SICStus Prolog one. The first one relies on the second one as explained in Section 2. Then, instead of adding a new module for the finite domain constraint library, we make available all the predefinitions in the module `plgenerated`, which is built by compiling the file `nadacflpfd.toy` joined with `basic.toy`, as shown in Figure 5. `nadacflpfd.toy` includes the file `cflpfd.toy` by the directive `include`; in fact, it is the only code in `nadacflpfd.toy`.

The result of this compilation gives the file `nadacflpfd.pl`, which contains all the stuff needed for goal solving but the actual implementation of the behaviour for finite domain built-ins. This is because a $\mathcal{TOY}$ program is compiled in terms of $\mathcal{TOY}$ built-ins, and it is not possible to specify its Prolog translation at this level. So, function definitions in the file `cflpfd.toy` are void, such as `0 #> 0 = true`. After compilation, this is translated into Prolog as shown in Figure 5.

On the other hand, the actual implementation of finite domain built-ins are in the file `cflpfdfile.pl`, and its contents replace the void definitions in the file `nadacflpfd.pl`, which now builds the module for the $\mathcal{FD}$ constraints library. In fact, it is the only module which is modified by loading this library, in contrast to the three or four modules (depending on the loaded libraries) of former cases.

Comparing this approach to the former one means that library developers have only to focus on modifying just two files: `cflpfd.toy`, for declaring predefined types, data constructors and function types; and `cflpfdfile.pl`, for implementing the actual behaviour of predefined functions. In the former approach, the library developer has to define the built-ins in a $\mathcal{TOY}$ file, compile this file, extract type definitions for constructors and functions, add them to `primFunct`*`Library`*`.pl`

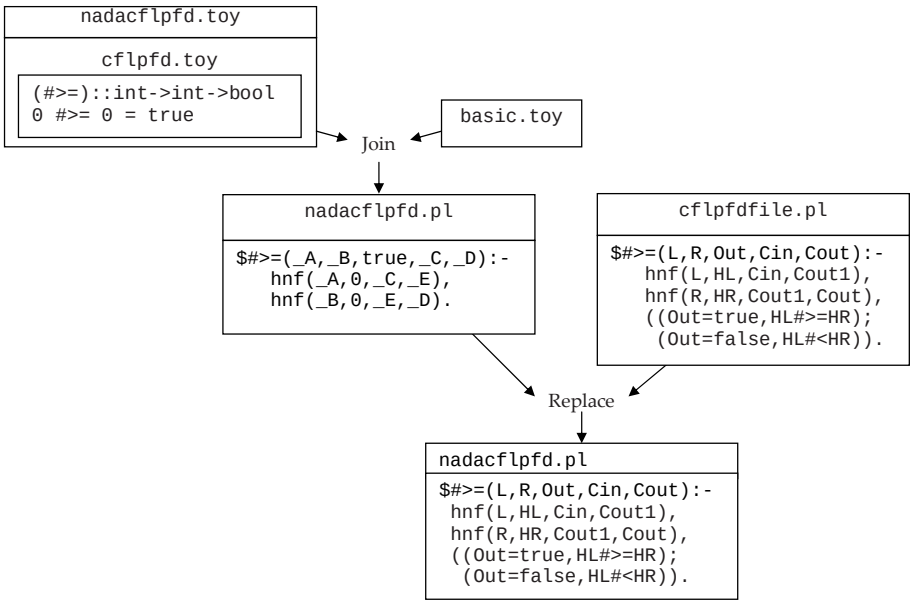

Fig. 5. Assembling Finite Domain Built-Ins into the Module `plgenerated`

and `basic`*`Library`*`.pl` in different formats, and modify the module interface for `primitivCod`*`Library`*`.pl`. Clearly, this process, besides tedious, is prone to errors.

5 Implementing Solver Cooperation

This section shows the implementation of the basic mechanisms for solver cooperation: binding and propagation. Binding is implemented with the new communication constraint `bridge`, whereas propagation is implemented by modifying the code of predefined constraints, allowing to send mate constraints to mate solvers [14]. Solver cooperation is allowed with binding alone or both binding and propagation. This allows to analyze the trade-off between communication flow and performace gain and decide the best option for a given program. Next, we show the implementation of binding and propagation (from Solver$^{\mathcal{FD}}$ to Solver$^{\mathcal{FD}}$ and viceversa).

5.1 Binding

The Prolog implementation of the bridge constraint is similar in nature to the translation of any defined function. In particular, the name of the function is preceded by the symbol `$` in order to avoid the name clash problem. The code excerpt below (in `cflpfdfile.pl`) shows its basic implementation (without obvious optimizations):

```
(1) $#==(L, R, Out, Cin, Cout):-
(2)  hnf(L, HL, Cin, Cout1), hnf(R,HR, Cout1, Cout2),
(3)  ((Out=true, Cout = ['#=='(HL,HR)|Cout2],
(4)    freeze(HL, HR is float(HL)), freeze(HR, HL is integer(HR)));
(5)   (Out=false, Cout = ['#/=='(HL,HR)|Cout2],
(6)    freeze(HL, (F is float(HL), {HR =\= F})),
(7)    freeze(HR, (0.0 is float_fractional_part(HR) -> (I is integer(HR), HL #\= I); true)))).
```

As the 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 constraint store and the outcoming constraint store. This store (in the following, mixed store) includes constraints from both domains $\mathcal{H}$ and $\mathcal{M}$, i.e., disequality constraints for constructed terms and equality and disequality bridges (in this paper, we have introduced $equiv\ e_1\ e_2 \rightarrow! \ false$, abbreviated as `#/==`, as the counterpart of $equiv\ e_1\ e_2 \rightarrow! \ true$ [14], abbreviated as `#==`). Notice that there is no need of explicitly account for equality constraints on $\mathcal{H}$ since they are handled by unification, whereas equality bridges should be explictly accounted since they cannot be implicitly handled because of the different related types (integers vs. reals). The mixed store also includes totality constraints [7]. Therefore, it includes the stores H, M, and T (this last one for the totality constraints). We have relied in the current predicate prototypes instead of using new arguments for different stores for the sake of rapid prototyping.

Line `(2)` flattens both `L` and `R` by calculating their hnfs, which always delivers either a variable or a number, therefore ensuring that no suspensions will occur from line `(3)` 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 calculation may involve during narrowing new $\mathcal{H}$ disequality constraints that have to be added to the mixed store.

A common use of the bridge constraint is as an argument of a goal connective (e.g., `f X Y = true <= X #> 0, X #== Y, Y < log 100`), but it also accepts its reification (e.g., `f X Y = B <= X #> 0, (X #== Y) == B, Y < log 2 100`). This means that if the value for `B` is `true`, then the constraint `X #== Y` is posted to the store M (line `(5)`), whereas if the value is `false`, then the complementary constraint (`X #/== Y`) is otherwise posted (line `(5)`).

Implementing both $equiv\ e_1\ e_2 \rightarrow! \ true$ and $equiv\ e_1\ e_2 \rightarrow! \ false$ 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 (performed by the operations `float` and `integer` in line `(4)`). Binding and matching are accomplished by unification. This constraint also amounts to an integral constraint over its right argument. For the second case (`#/==`), we have to state in Solver$^{\mathcal{M}}$ that both arguments are not equal, which cannot be directly handled by SICStus solvers, as before. So, whenever an argument becomes (or is) ground in a domain $\mathcal{D}$, then a disequality constraint between the casted ground variable and its mate variable can be posted to the underlying solver for the mate domain $\mathcal{D}'$ (lines `6-7`).

5.2 Propagation: Solver$^{\mathcal{FD}}$ to Solver$^{\mathcal{R}}$

Propagating a constraint in $\mathcal{FD}$ to $\mathcal{R}$ is possible if: the user has enabled propagation with the command `/prop`, there is a bridge relating any of the involved variables, and the constraint is allowed to be propagated (cfr. [14]). The propagation amounts to, first, send a mate constraint from $\mathcal{R}$ to $\mathcal{FD}$, and, second, create bridges for the rest of variables in the $\mathcal{FD}$ constraint which are not involved in bridges, therefore creating new $\mathcal{R}$ variables with integral values which may be further related in other $\mathcal{FD}$ constraints. The code excerpt below (which can be found in `cflpfdfile.pl`)

shows its basic implementation (without considering obvious optimizations) for a concrete constraint `#>`:

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

This predicate follows the same prototype (line `(1)`) as `#==`, since it is a binary relation which can be reified. Its two input arguments (`L` and `R`) are demanded to be in hnf (line `(2)`), and a primitive constraint is posted to the underlying $\mathcal{FD}$ solver, depending on the Boolean result of the function (line `(3)`). If propagation is active (indicated by the dynamic predicate `prop_active` in line `(4)`), then bridges relating the arguments of `#>` are looked for in the mixed store in order to find mate variables in $\mathcal{R}$ (lines `(5)-(6)`). 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 `(7)-(8)`).

5.3 Propagation: Solver$^{\mathcal{R}}$ to Solver$^{\mathcal{FD}}$

Propagating a constraint in $\mathcal{R}$ to $\mathcal{FD}$ is possible in the same conditions stated in the previous section. The propagation amounts to send mate constraints as before, but bridges for the rest of variables in the $\mathcal{R}$ constraint are not created since their integral nature is not for sure. The code excerpt below (which can be found in `primitiveCodClpr.pl`) shows its basic implementation (without considering obvious optimizations) for a concrete constraint `>`:

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

After analogous steps to the previous subsection (lines `(1)-(3)`), the next two lines deal with the explicit interaction between Solver$^{\mathcal{H}}$ and Solver$^{\mathcal{R}}$ [32]. Whenever a disequality constraint occurs, it is assumed to involve terms in $\mathcal{H}$ (the type of a variable is not always known because types are checked and inferred at compile-time but this information is not present at run-time) so that it is sent to the mixed store (this task is performed in the evaluation of the disequality constraint with the predicate `notEqual` [4]). Therefore, we need to check whether the disequality affect a real variable; if so, the constraint is sent to the underlying solver for reals and removed from the mixed store (lines `(4)-(5)`). Next, if propagation is active, a similar procedure to the one performed for the propagation in the other direction follows. However, notice that there are more possibilities for sending a mate constraint to Solver$^{\mathcal{FD}}$ (see Table 4 in [14]), depending on whether bridges are found

for the related variables (a `true` value for `BL` (resp. `BR`) means that a bridge relating the variable in the left(resp. right)-hand-side of the constraint has been found).

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 software architecture, the data-flow program compilation process, library loading, 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 [23].

More specifically, we have described the internal communication between Solver$^{\mathcal{R}}$ and Solver$^{\mathcal{FD}}$ via binding and propagation. We have sketched their implementation, and shown that binding manages 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 that 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 [14]. The descriptions in this paper might reveal useful for implementors of other related declarative systems.

In general, solver cooperation have 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) [6], Prolog III [10] and Prolog IV [31] allow solver cooperation, mainly limited to Booleans, reals and naturals. Also, the language NCL [35] 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 [35]. 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 nature to all those systems as its implementation follows the theoretical principles recently described in [14]. 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, [22] 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, [30] 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 num-

bers) 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 to 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 [29], 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`, which allows the dynamic integration of arbitrary external (standalone) solvers to enable the collaborative processing of constraints [16,17,18]. The similarities between $\mathcal{TOY}$ and `Meta-S` are not coming from the implementation point of view but because solver cooperation in `Meta-S` was implemented following [25]. Such a proposal keeps some similarities with ours, although there are evident differences identified in [14]. As future work, we plan to develop a comparison between $\mathcal{TOY}$ and the system `Meta-S` with respect to solver cooperation (i.e., implementation issues and performance analysis), and to integrate another solvers.

References

[1] Abengózar-Carneros et al, M., *TOY: A Multiparadigm Declarative Language. Version 1.0*, Technical Report SIP-119/00, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Programación, UCM (2002).

[2] Antoy, S., *Definitional Trees*, in: *3rd International Conference on Algebraic and Logic Programming (ALP'92)*, number 632 in LNCS (1992), pp. 143–157.

[3] Antoy, S. and M. Hanus, *Compiling Multi-Paradigm Declarative Programs into Prolog*, in: *Frontiers of Combining Systems*, 2000, pp. 171–185.

[4] Arenas, P., A. Fernández, A. Gil, F. López-Fraguas, M. Rodríguez-Artalejo and F. Sáenz-Pérez, $\mathcal{TOY}$. *A Multiparadigm Declarative Language. Version 2.2.3* (2006), R. Caballero and J. Sánchez (Eds.), Available at `http://toy.sourceforge.net`.

[5] Arenas, P., A. Gil and F. López-Fraguas, *Combining Lazy Narrowing with Disequality Constraints* (1994).

[6] Benhamou, F. and W. Older, *Applying Interval Arithmetic to Real, Integer and Boolean Constraints*, The Journal of Logic Programming **32** (1997), pp. 1–24.

[7] 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* (2005), pp. 8–13.

[8] Caballero, R. and F. J. López-Fraguas, *Dynamic-Cut with Definitional Trees*, in: *FLOPS '02: Proceedings of the 6th International Symposium on Functional and Logic Programming* (2002), pp. 245–258.

[9] Carlsson, M., G. Ottosson and B. Carlson, *An Open-ended Finite Domain Constraint Solver*, in: U. Montanari and F. Rossi, editors, *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, number 1292 in LNCS (1997), pp. 191–206.

[10] Colmerauer, A., *An Introduction to PROLOG III*, Communications of the ACM (CACM) **33** (1990), pp. 69–90.

[11] de la Banda, M. J. G., B. Demoen, K. Marriott and P. J. Stuckey, *To the Gates of HAL: A HAL Tutorial*, in: Z. Hu and M. Rodríguez-Artalejo, editors, *6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, Lecture Notes in Computer Science **2441** (2002), pp. 47–66.

[12] del Vado-Vírseda, R., *A Demand-driven Narrowing Calculus with Overlapping Definitional Trees*, in: *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03)* (2003), pp. 213–227.

[13] Diaz, D. and P. Codognet, *Design and Implementation of the GNU Prolog System*, Journal of Functional and Logic Programming **2001** (2001).

[14] Estévez-Martín, S., A. Fernández, T. Hortalá-González, M. Rodríguez-Artalejo, F. Sáenz-Pérez and R. del Vado-Vírseda, *A Proposal for the Cooperation of Solvers in Constraint Functional Logic Programming*, ENTCS (2006), in Press.

[15] Fernández, A., T. Hortalá-González, F. Sáenz-Pérez and R. del Vado-Vírseda, *Constraint Functional Logic Programming over Finite Domains*, ENTCS (2006), in Press.

[16] Frank, S., P. Hofstedt and P. R. Mai, *A Flexible Meta-solver Framework for Constraint Solver Collaboration*, in: A. Günter, R. Kruse and B. Neumann, editors, *26th Annual German Conference on AI, Advances in Artificial Intelligence (KI 2003)*, Lecture Notes in Computer Science **2821** (2003), pp. 520–534.

[17] Frank, S., P. Hofstedt and P. R. Mai, *Meta-S: A Strategy-Oriented Meta-Solver Framework*, in: I. Russell and S. M. Haller, editors, *Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS Conference)* (2003), pp. 177–181.

[18] Frank, S., P. Hofstedt and D. Reckman, *Meta-S - Combining Solver Cooperation and Programming Languages*, in: A. Wolf, T. W. Frühwirth and M. Meister, editors, *19th Workshop on (Constraint) Logic Programming (W(C)LP)*, Ulmer Informatik-Berichte **2005-01** (2005), pp. 159–162.

[19] Frühwirth, T., *Theory and Practice of Constraint Handling Rules*, The Journal of Logic Programming **37** (1998), pp. 95–138.

[20] González-Moreno, J., M. Hortalá-González, F. López-Fraguas and M. Rodríguez-Artalejo, *An Approach to Declarative Programming Based on a Rewriting Logic*, The Journal of Logic Programming **40** (1999), pp. 47–87.

[21] González-Moreno, J., M. Hortalá-González and M. Rodríguez-Artalejo, *A Higher Order Rewriting Logic for Functional Logic Programming*, in: *14th International Conference on Logic Programming (ICLP'97)* (1997), pp. 153–167.

[22] Goualard, F., *Component Programming and Interoperatibility in Constraint Solver Design*, in: K.Apt, R. Barták, E. Monfroy and F. Rossi, editors, *ERCIM Workshop on Constraints* (2001).

[23] Granvilliers, L., E. Monfroy and F. Benhamou, *Cooperative Solvers in Constraint Programming: a Short Introduction*, ALP Newsletter **14** (2001).

[24] Hanus, M., *Curry: a Truly Integrated Functional Logic Language* (1999), http://www.informatik.uni-kiel.de/~curry/.

[25] Hofstedt, P. and P. Pepper, *Integration of Declarative and Constraint Programming*, Theory and Practice of Logic Programming (2006), in Press.

[26] Hortalá-González, T., F. López-Fraguas, J. Sánchez-Hernández and E. Ullán-Hernández, *Declarative Programming with Real Constraints* (1997).

[27] Loogen, R., F. López-Fraguas and M. Rodríguez-Artalejo, *A Demand Driven Computation Strategy for Lazy Narrowing*, in: *PLILP*, 1993, pp. 184–200.

[28] López-Fraguas, F. and J. Sánchez-Hernández, *TOY: A Multiparadigm Declarative System*, in: P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, number 1631 in LNCS (1999), pp. 244–247.

[29] Monfroy, E., "Solver Collaboration for Constraint Logic Programming," Ph.D. thesis, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (1996).

[30] Monfroy, E., M. Rusinowitch and R. Schott, *Implementing Non-Linear Constraints with Cooperative Solvers*, Research Report 2747, Centre de Recherche en Informatique de Nancy, INRIA-Lorraine (1995).

[31] 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)* (1997), pp. 235–238.

[32] Sánchez-Hernández, J., *TOY: Un Lenguaje Lógico funcional con restricciones* (1998), available (in Spanish) at `http://babel.dacya.ucm.es/jaime/publications.html`.

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

[34] Van Roy, P., P. Brand, D. Duchier, S. Haridi, M. Henz and C. Schulte, *Logic Programming in the Context of Multiparadigm Programming: the Oz Experience*, Theory and Practice of Logic Programming **3** (2003), pp. 717–763.

[35] Zhou, J., *Introduction to the Constraint Language NCL*, The Journal of Logic Programming **45** (2000), pp. 71–103.