Interfacing a Functional Logic Language with a Finite Domain Solver

Teresa Hortalá-González, Fernando Sáenz-Pérez *

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
{teresa,fernan}@sip.ucm.es

Abstract. In this paper, we present a straightforward way for interfacing the functional logic language $\mathcal{TOY}$ with a finite domain solver. Since $\mathcal{TOY}$ programs are compiled to Sicstus Prolog programs, we use the Sicstus' finite domain library to allow the expression of a finite domain problem in $\mathcal{TOY}$. Finite domain $\mathcal{TOY}$ programs consist of functional logic $\mathcal{TOY}$ rules interfaced with constraint Sicstus clauses. This approach allows us to take advantage of the full functionality of Sicstus Prolog constraints.

Keywords: Functional Logic Programming, Finite Domain Constraints.

1 Introduction

Declarative programming is intended to separate the problem formulation from the procedure to solve the problem itself. With logic programming, one can express the problem in first order predicate calculus. Functional programming allows to express problems in terms of higher order functions. In turn, constraint programming (see [Smi95] for a tutorial) entails a new step in declarative programming by allowing to express constrained optimization problems (COPs) as unknowns (variables with an associated domain), and constraints which must be satisfied by all such variables. Depending on the chosen variable domain for constraint programming, expressiveness is guided to different problem domains. Real-life problems to be typically solved with constraint programming include planning, scheduling, and resource allocation. Mathematical programming (see [Fou99] for a survey) can be seen as a classical approach to solve problems expressed with disequations of variables in a real domain or a combination of the real and integer domains. It has proven both useful and capable to cope with complex problems but it suffers from a lack of expressiveness which can be dealt with constraint programming (for instance, non-linear constraints, which prevent the use of the fast solving methods based on the simplex and branch and bound methods). In addition, using integer non-binary variables imply a performance

* This work has been supported by the Spanish CICYT project TIC98-0445-C03-02 ("TREND").

degradation that means a tradeoff between expressiveness and efficiency (non-binary problems can be coded with binary variables at the cost of an awkward problem reformulation in general) typically used in mathematical programming. On the other hand, constraint programming allows a clear expression of non-linear constraints. Interval reasoning is used for handling non-linear constraints on real variables in constraint systems, which means a poor efficiency. For some complex applications in an integer domain, constraint programming has been proven to outperform integer programming [DL+97,Smi96,JD95]. Current research aims to the integration of the techniques inherited from both constraint and mathematical programming.

Functional logic programming (FLP), in turn, aims to integrate functional and logic programming, allowing the use of techniques from both paradigms into the same declarative framework (see [Han94] for a survey). Moreover, the combination of ideas of the two worlds gives rise to new features specific to FLP. This work is a contribution for further augmenting the expressive power of FLP by adding the possibility of solving finite domain (FD) constraint problems in the context of the functional logic programming language $\mathcal{TOY}$ [LS99],[Rod01], [AA+01]. We use the Sicstus' finite domain library to allow the expression of a finite domain problem with Prolog predicates which are interfaced with $\mathcal{TOY}$programs. This allows to express finite domain problems with predicates which are handled from a $\mathcal{TOY}$ program.

There has been a huge work in developing the constraint logic programming paradigm (see [JM94] for a survey), which replaces unification by constraint satisfaction. General frameworks for constraint functional logic programming (CFLP) are proposed in [FH99,Lop92], but, as expected, implementations suffer from a lack of efficiency [Fer00,FH99]. Oz [Smo95] is a functional logic language based on concurrent constraint solving which adds the concept of state (against a pure functional logic programming language) by means of the object oriented paradigm. Curry [Han00] integrates features from functional languages, logic languages and concurrent programming. In addition, there are other related works that embody real constraints in functional logic languages. [AH+96,LS99] includes linear constraints over real numbers and disequality constraints between syntactic terms that works together. [Lux01] describes the addition of linear constraints over real numbers to Curry and also optimizing constraints. It should also be noted that non-linear constraints are also managed by the last two approaches, by delaying non-linear terms until they become linear. In spite of these works on adding linear constraints to a functional logic language, it should be mentioned that, yet their own importance, real life problems do require fast solving methods, and consequently classical approaches have been notably upgraded as [BF+00] surveys. This is to say that not only expressiveness is important, but also efficiency.

The rest of the paper is organized as follows. Section 2 introduces the constraint functional logic language we use as a host language for FD constraint solving. Section 3 introduces general concepts about constraint programming. Section 4 shows the way FD constraints have been interfaced to $\mathcal{TOY}$ programs.

Section 5 briefly describes the implementation of the proposed approach. Finally, section 6 summarizes some conclusions and points out future work.

2 The $\mathcal{TOY}$ Language

$\mathcal{TOY}$ is a purely declarative, constraint functional logic language, which is typed, lazy, and higher order, whose solid foundations can be found in [LLR93], [GH+99], [GHR97]. We present here the subset of the language relevant to this work (see [AA+01] for a more complete description and a number of representative examples).

We will use constructor-based signatures $\Sigma = \langle DC, FS \rangle$, where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ resp. $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are respectively sets of *data constructors* and *defined function symbols* with associated arities. As notational conventions, we will assume $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$. We also assume that many countable variables (noted as X, Y, Z, etc.) are available. Given any set $\mathcal{X}$ of *variables*, we will consider the set $Exp_\Sigma(\mathcal{X})$ of all terms built from symbols in $\mathcal{X} \cup DC \cup FS$, and also the set $Term_\Sigma(\mathcal{X})$ of all terms built from symbols in $\mathcal{X} \cup DC$. Terms $l, r, e \in Exp_\Sigma(\mathcal{X})$ will be called *expressions*, while terms $s, t \in Term_\Sigma(\mathcal{X})$ will be called *constructor terms* or also *data terms*. Expressions without variables will be called *ground* or *closed*. Moreover, we will say that an expression e is in *head normal form* iff e is a variable X or has the form $c(\overline{e}_n)$ for some data constructor $c \in DC^n$ and some n-tuple of expressions $\overline{e}_n$.

A $\mathcal{TOY}$ program consists of *datatype*, *type alias* and *infix operator* definitions, and rules for defining *functions*. Its syntax is mostly borrowed from Haskell [HAS97], with the remarkable exception that variables begin with upper-case letters whereas constructor symbols use lower-case, as function symbols do. In particular, functions are *curried* and the usual conventions about associativity of application hold.

Datatype definitions like data nat = zero | suc nat, define new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type.

Types $\tau, \tau', \ldots$ can be constructed types, tuples $(\tau_1, \ldots, \tau_n)$, or functional types of the form $\tau \to \tau'$. As usual, $\to$ associates to the right. $\mathcal{TOY}$ provides predefined types such as [A] (the type of polymorphic lists, for which Prolog notation is used), bool (with constants true and false), int for integer numbers, or char (with constants 'a','b', ...). *Type alias* definitions like type parser_rec A = [A] → [A] are also allowed. Type alias are simply abbreviations, but they are useful for writing more readable, self-documenting programs. Strings (for which we have the definition type string = [char]) can also be written with double quotes. For instance, "sugar" is the same as ['s','u','g','a','r'].

A $\mathcal{TOY}$ program defines a set FS of functions. Each $f \in FS^n$ has an associated principal type of the form $\tau_1 \to \ldots \to \tau_m \to \tau$ (where τ does not contain $\to$). Number m is called the *type arity* of f and well-typedness implies that $m \geq n$. As usual in functional programming, types are inferred and, optionally, can be declared in the program.

We distinguish two important syntactic domains: *patterns* and *expressions*. Patterns can be understood as denoting data values, i.e. values not subject to further evaluation, in contrast with expressions, which can be possibly reduced by means of the rules of the program. Patterns $t, s, \ldots$ are defined by $t ::= X \mid (t_1, \ldots, t_n) \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_n$, where $c \in DC^m$, $n \leq m$, $f \in FS^m$, $n < m$, and t_i are also patterns. Notice that partial applications (i.e., application to less arguments than indicated by the arity) of c and f are allowed as patterns, which is then called a *HO pattern*, because they have a functional type. Therefore function symbols, when partially applied, behave as data constructors. HO patterns can be manipulated as any other patterns; in particular, they can be used for matching or checked for equality. With this *intensional* point of view, functions become 'first-class citizens' in a stronger sense that in the case of 'classical' FP.

Expressions are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e_1\ e_2)$, where $c \in DC$, $f \in FS$, and e_i are also expressions. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore $e\ e_1 \ldots e_n$ is the same as $(\ldots(\,(e\ e_1)\ e_2\,)\ldots)\,e_n)$. Of course, expressions are assumed to be well-typed. *First order patterns* are a special kind of expressions which can be understood as denoting data values, i.e. values not subject to further evaluation, in contrast with expressions, which can be possibly reduced by means of the rules of the program.

Each function $f \in FS^n$ is defined by a set of conditional rules of the form

$$f\ t_1 \ldots t_n\ = e \ \Longleftarrow\ l_1 == r_1, \ldots, l_k == r_k$$

where $(t_1 \ldots t_n)$ form a tuple of linear (i.e., with no repeated variable) *patterns*, and e, l_i, r_i are *expressions*. No other conditions (except well-typedness) are imposed to function definitions. Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to e if all the conditions $l_1 == r_1, \ldots, l_k == r_k$ are satisfied. The condition part is omitted if $k = 0$.

The symbol $==$ stands for *strict equality*, which is the suitable notion (see e.g. [Han94]) for equality when non-strict functions are considered. With this notion, a condition `e == e'` can be read as: `e` and `e'` can be reduced to the same pattern. When used in the condition of a rule, $==$ is better understood as a constraint (if it is not satisfyable, the computation fails), but the language contemplates also another use of $==$ as a function, returning the value `true` in the case described above, but `false` when a clash of constructors is detected while reducing both sides.

In addition to $==$, $\mathcal{TOY}$ incorporates other predefined functions like the arithmetic functions `+`,`*`, ..., or the functions `if_then` and `if_then_else`, for which the more usual infix syntax is allowed. Symbols `==`,`+`,`*` are all examples of *infix operators*. New operators can be defined in $\mathcal{TOY}$ by means of *infix* declarations, like `infixr 50 ++` which introduces `++` (used for list concatenation, with standard definition) as a right associative operator with priority 50. Operators for data constructors must begin with ':', as in the declaration `infix 40 :=`. *Sections*, or partial applications of infix operators, as `(==3)` or `(3==)` are also allowed.

A distinguished feature of $\mathcal{TOY}$ is that no confluence properties are required for the programs, and therefore functions can be *non-deterministic*, i.e. return several values for given (even ground) arguments. For example, the rules `coin = 0` and `coin = 1` constitute a valid definition for the 0-ary non-deterministic function `coin`. Two reductions of `coin` are allowed, which lead to the values `0` and `1`. The system try in the first place the first rule, but, if backtracking is required by a later failure or by request of the user, the second rule is tried. Another way of introducing non-determinism in the definition of a function is by adding *extra* variables in the right side of the rules, as in `z_list = [0|L]`. Any list of integers starting by 0 is a possible value of `z_list`. Anyway, note that in this case only one reduction is possible.

Disequality constraints over arbitrary free data types and arithmetic constraints over the real numbers are already available in the $\mathcal{TOY}$ system. Implementing disequality constraints requires to maintain a constraint store with constraints in solved form X `/=` t, which must be awaken whenever X becomes bound. A first implementation technique for a subset of CFLP($\mathcal{H}$), consisting of so-called *uniform programs*, was presented in [KLMR92]. Later on, an implementation of full CFLP($\mathcal{H}$), based on a translation into Prolog with demand driven strategy, was proposed in [AGL94]. [SL98] provides a semantics (Goal Oriented Rewriting Calculus with disequalities) for $\mathcal{TOY}$ which closely approximates the current implementation, which is based on the more concrete approach CRLW which in turn departs from [GH+99]. The current $\mathcal{TOY}$ system relies on this technique to solve disequality constraints, and it invokes the CLP($\mathcal{R}$) solver of Sicstus Prolog [Sic99] to solve real arithmetic constraints. Of course, user-defined functions occurring within arithmetic constraints have to be evaluated before invoking the solver. The $\mathcal{TOY}$ system has two modes of use: with or without the arithmetic solver. The user can switch between these two modes by means of special commands. Activating the arithmetic solver causes some overhead in those computations that actually do not need it.

3 Constraint Programming

There are two isolated steps in using constraint programming to solve a FD problem. First, the problem representation, which involves the declaration of variables and constraints, and possibly the declaration of a search method. This step is specific to the problem domain and needs a language to both declare variables and post constraints on them. This language may be declarative, or even imperative, such as C++ [Pug94], although a more natural description is found in the former. The second step is the solution search, which consists of finding a suitable assignment for each variable such that all constraints are simultaneously satisfied, and, in addition, allowing to optimize a function. This is expressed as a cost function which, evaluated over each solution to the problem, returns a value indicating the appropriateness of such solution under a given criterion. While the first step is intended to be performed by the programmer, the second

step is performed by the underlying constraint solving system[1]. The first step, i.e., modeling, is quite important in solving the problem so the responsibility of building efficient models is led to the programmer. Several semantically equivalent models may yield great differences in efficiency (symmetry breaking and other techniques aims to build more efficient models [SSW99]).

A constraint optimization problem involving finite domains can be stated in the following way. Given a tuple (V, D, C, f), where:

- $V \in V_1, ..., V_n$, a set of domain variables,
- $D \in D_1, ..., D_n$, a set of finite integers domains,
- $C \in C_1, ..., C_m$, a set of constraints between the variables in V,
- $f(V)$, an objective function,

the goal is to find an assignment of a value from D_i to each variable V_i which satisfies the constraints in C and optimizes (maximizes or minimizes) the objective function f.

Solving in constraint systems is based on constraint propagation and labeling. The first prunes the search space by reducing domains, and the second finds solutions by assigning values to variables.

The system starts solving by propagating the effects of the constraints over the domains of variables. This means that, in general, propagation implies that each current domain will decrease its cardinality (pruning). There are several propagation algorithms in the literature [Tsa93] which behave differently and may reach different fixed points. The fixed point is reached whenever there is no further domain reduction. These algorithms implement an iterative procedure which looks for a stable situation (fixed point), or a failure (a domain becomes empty, i.e., there is no possibility of finding an assignment for the related variable such that all the constraints are satisfied). Finding a fixed point with non-singleton domains does not mean that there are definitely multiple solutions to the problem, and it does not even ensure that at least one solution exists. Propagation is not complete in the sense of ensuring the existence of solutions. Instead, it is used to find what assignments definitely does not lead to a solution. The premise in this approach is to identify in advance, as soon as possible, what partial solution (where not all domains are singletons) is not a solution before trying to assign all the variables. Note that this is an approach different from enumeration techniques, which try to find solutions by simultaneously assigning values to all the variables, so that one knows that a solution is when all variables have been assigned.

Once propagation procedure reaches a fixed point and at least one domain is not a singleton, labeling is initiated in order to find feasible assignments. Indeed, the search for solutions could be seen at this point from an enumeration point of view. However, each time a variable is assigned to a value, propagation can be started until a fixed point had been reached. Next, a new assignment can be

[1] Nonetheless, it is possible for the programmer to provide a search procedure specialized with the problem knowledge for labeling, therefore being hopefully more efficient than the predefined procedure(s) given by the underlying system.

made, a new propagation cycle started, and so on, until a solution is computed or not found. The latter means that backtracking must be started in order to find another possible assignment. Each time a variable is labeled (assigned to a value among the possible values in its domain), a choice point must be annotated in order to try different assignments through backtracking.

Our experience shows that solving real-life complex problems implies an orthogonal way in using constraints in a host language. Several (sequential) programming sections can be identified: input data preparation, variable declaration, constraint posting (and optionally search procedure tuning), and data output. This suggested us that one important point in solving real-life constraint problems is to provide the host language (the functional logic language $\mathcal{TOY}$) with a connection to a finite domain constraint solver. The objective is to allow the programmer to prepare the input data in the host language, then to express the constraint problem in the target constraint system, and finally to get the result (output data) for continuing the computation. This orthogonal way of problem formulation comes from the fact that a problem usually can be decomposed into several subproblems, which, because their nature, can be better solved with specific programming paradigms. For instance, it is acknowledged that combinatorial problems in general are better expressed and efficiently solved with constraint propagation than short-term planning, which is better solved with mathematical programming. In addition to this situation, there is the need of 'playing' with the solution reached for a COP in the host language in order to build decision support systems.

4 Interfacing $\mathcal{TOY}$ with the Sicstus Constraint Solver

Sicstus Prolog [Sic99] provides a constraint logic programming library over the finite domain of integer subsets which can be effectively used for building a functional logic constraint system. We use this library at Prolog level taking for granted that $\mathcal{TOY}$ programs are compiled to Prolog. Our goal in interfacing $\mathcal{TOY}$ with the Sicstus Constraint Solver is depicted in figure 1. The left side shows the composition of a generic FD problem which has to be interfaced to $\mathcal{TOY}$. Since $\mathcal{TOY}$ programs are compiled to Prolog, we propose to use the Sicstus Prolog framework to perform constraint solving in the same tier than the compiled $\mathcal{TOY}$ program will be run. This means that input parameters to the FD problem have to be fed to the Prolog specification, and, in turn, instantiated or constrained FD variables have to be returned to the calling function application. Since the finite domain is a subset of integers, we consider the interface by sending and receiving integers, so that FD functions have to be type checked for consistency. This approach can be glued with the formalization below considering that the semantics of an FD function can be extensionally specified taking advantage of non-determinism:

$$\begin{aligned} f\ p_1^1 \ldots p_n^1 &= (t_1^1, \ldots, t_m^1) \\ &\vdots \\ f\ p_1^k \ldots p_n^k &= (t_1^l, \ldots, t_m^l) \end{aligned}$$

where $p_i^j, t_i^j \in \mathbb{Z}$, and $n, m, l, k \in \mathbb{N}^+$. This formulation expresses that for a given pattern i of parameters $(p_1^i, \ldots, p_n^i)$, there is the set of solutions $\{(t_1^j, \ldots, t_m^j) : j \subset \{1, ldots, l\}\}$ for the optimization problem.

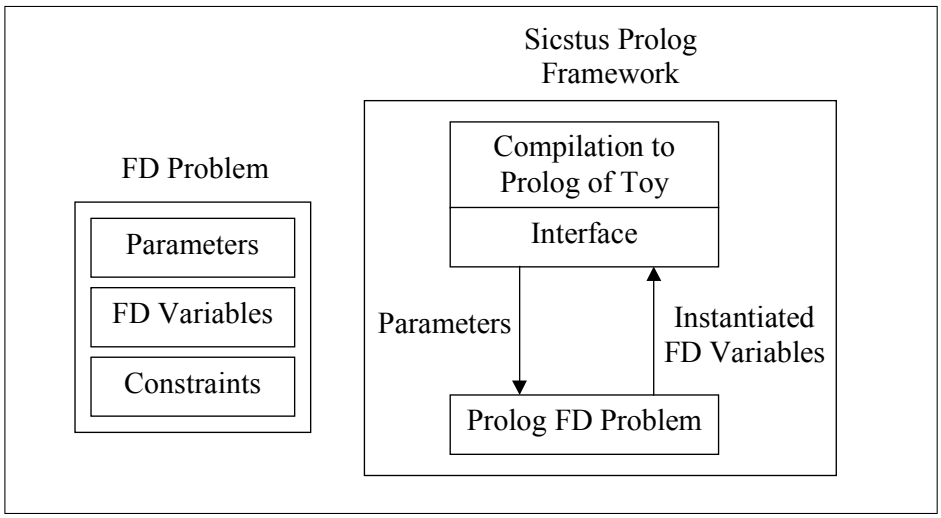

Fig. 1. Interfacing $\mathcal{TOY}$ with a Prolog FD Constraint Solver

A $\mathcal{TOY}$ program with calls to finite domain problem solving consists of, firstly, an include statement which loads the Prolog FD program, and, secondly, calls to the Prolog side by means of the $\mathcal{TOY}$ function `evalfd`, whose prototype is `evalfd :: [char] -> [int] -> [int]`. This prototype states that `evalfd` has two arguments: the name of the Prolog predicate to be called (a string), and the (integer) input parameters; in addition, it returns a list of integer variables which may be possibly bound via labeling.

The next code fragment shows a possible use of the proposal, where the primitive `includeclpfd` includes the Prolog program `queens.pl` (the well-known queens problem formulated with constraints), and the function `append` defines list concatenation.

```
includeclpfd "queens.pl"

append :: [A] -> [A] -> [A]
append [] Ys = Ys
append [X|Xs] Ys = [X|append Xs Ys]
```

With this code loaded and compiled in the $\mathcal{TOY}$ system, one can submit the goal `evalfd "queens" [4] == L`, which returns as possible solutions `L == [2,4,1,3]`, and `L == [3,1,4,2]`. Another goal is `evalfd "queens" [1+3] ==`

`(append [2,4] [1,3])`, which returns success. `evalfd` can also be used to determine the input parameter, as in `evalfd "queens" [X] == [2,4,1,3]`, which returns `X == 4`, or even `evalfd "queens" [X] == [2,X|Y]`, which returns all solutions starting with 2 and at least two elements in the list, as `X == 4`, `Y == [1,3]`, and `X == 5`, `Y == [3,1,4]`.

Finite domain variables are constrained after the finite domain call even they are not bound by labeling. For instance, we can submit the goal `evalfd "queens" [X] == [A,B,C,D], append [2] [4,1,3] == [A,B,C,D]`, which returns `X == 4`, `A == 2`, `B == 4`, `C == 1`, `D == 3` .

Additionally, we can have a higher order use of the `evalfd` function, as in `F == evalfd "queens", F [4] == L`, which returns `L == [2,4,1,3]`, and `L == [3,1,4,2]` both with `F == (evalfd "queens")`.

The presented approach is not only valid for interfacing FD predicates with $\mathcal{TOY}$, but also for any other Prolog predicate obeying the interface restrictions imposed by `evalfd`. This technique could be extended for defining a general interface with Prolog predicates.

5 Implementation

The implementation follows a compilation of $\mathcal{TOY}$ programs to Sicstus programs. A $\mathcal{TOY}$ program is implemented as a set of Sicstus Prolog clauses, which is a classical compilation-to-Prolog approach for implementing narrowing. Variables and expressions are represented via Prolog variables and terms, respectively. $\mathcal{TOY}$ higher order programs are firstly translated into first order programs following [Gon93]. Equality and disequality constraints are also represented by terms following an infix representation for operators '==' and '/='. In this approach, we can take advantage of Prolog's unification, saving some work which the Sicstus underlying system efficiently performs. However, due to the existence of disequality constraints, Prolog unification is not enough to represent $\mathcal{TOY}$ unification, and it is needed to maintain a (disequality) constraint store which allows us to detect whether two variables can be unified in a Prolog way.

In order to commit to the conditions imposed in section 4, the user has to use a directive in order to load the FD Prolog module that defines the FD problem. The type checking stage ensures the conditions on parameters of an FD call as well as its outcome.

5.1 Computing Head Normal Forms

A head normal form (HNF) is any language expression which is not a total function application, i.e., a variable or expression which starts with a constructor. Note that a partial application $f\ t_1 \ldots t_m$, $f\ \in FS^n$, $m < n$, is regarded as a constructor symbol in $\mathcal{TOY}$. Hence, $ft_1 \ldots t_m$ represents a pattern and is in HNF already. HNFs are only computed when they are demanded following the lazy narrowing approach. A HNF is computed as shown in the following code excerpt:

```
hnf(E,H,Cin,Cout) :-
  var(E), !,
  (
     var(H),!,H=E,Cin=Cout
  ;
     extractCtr(E,Cin,Cout1,CE),H=E,
     propagate(H,CE,Cout1,Cout)
   ).
hnf(susp(Fun,Args,R,S),H,Cin,Cout) :-
  !,
  (
     S==hnf,!,hnf(R,H,Cin,Cout)
  ;
     H=R,S=hnf,hnf_susp(Fun,Args,H,Cin,Cout)
   ).
hnf(T,H,Cin,Cin) :- H=T.
```

The three clauses above correspond to variable, function application (suspension), and terms starting with a constructor symbol, respectively. The first argument is the syntactic object which has to be translated; the second argument is its translation to HNF; the third and fourth arguments represent the constraint store before and after the translation, respectively. The predicate `extractCtr` extracts the disequality constraints related to the syntactic object to be translated. The predicate `propagate` transforms constraints to solved forms by taking into account the information in the constraint store. The term `susp` represent a suspension `Fun` with arguments `Args` which may be evaluated to `R`, indicated by its status `S`. This term is used in the second clause for identifying whether the function is a suspension or not. The call to `hnf` is somewhat subtle in the branch starting with `S==hnf` in the second clause. Instead of just doing $R = H$, we have to take into account the constraint store looking for disequalities which can avoid the unification of `R` (in HNF already) with `H`, since `H` can be partially instantiated in a previous computation step. The call to `hnf` is simply a short way to commit to the disequalities present in the store.

5.2 Strict Equality

A naïve strict equality implementation is as follows:

```
equal(X,Y,Cin,Cout) :-
  hnf(X,HX,Cin,Cout1),
  hnf(Y,HY,Cout1,Cout2),
  equalHnf(HX,XY,Cout2,Cout).
```

But another more precise approach is followed in order to identify the different syntactic objects as soon as possible, therefore reducing the search space (see [SL98] for details).

5.3 Rule Application

Demand driven strategy implemented by $\mathcal{TOY}$ follows [LLR93] which in turn uses definitional trees [Ant92]. Details of implementation can be found in [SL98]. However, we are interested in rule application for FD functions, which has a specific treatment due to the conditions which must hold. In particular, arguments have to be integers, as well as the result, so that head normal forms are computed for input parameters. For instance, a finite domain function translation could be:

```
evalfd(CL, PL, H, Cin, Cout):-
      nf(CL, HCL, Cin, Cout1),
      hnf(PL, HPL, Cout1, Cout),
      toyListToIntPrologList(HPL,PrologList),
      toyStringToProlog(HCL,PrologString),
      name(Predicate, PrologString),
      FDGoal =.. [Predicate,PrologList,Result],
      call(FDGoal),
      intPrologListToToyList(Result,H).
```

First, the Prolog predicate name `CL` is required to be in normal form, and parameters `PL` are required to be in HNF. Second, some translations needed to convert Prolog and $\mathcal{TOY}$ representations are performed. Next, the call to the Prolog predicate is built by adding the list of input parameters and the list of output variables as arguments of the finite domain predicate. The Prolog FD goal is then called, and, finally, the $\mathcal{TOY}$ representation for the result is built. This translation assumes that the finite domain predicate has arity 2, its first argument is the input parameter list, and the second one is the output finite domain variables list. `Result` is bound by the FD predicate in the Prolog side specification of the FD problem.

6 Conclusions and Future Work

In this work we have shown a straightforward way of interfacing the functional logic language $\mathcal{TOY}$ with the finite domain solver of Sicstus Prolog, by taking advantage of the Prolog compilation approach for $\mathcal{TOY}$ programs. This allows us to orthogonally express constraint optimization problems with $\mathcal{TOY}$ as a host language, and to take advantage of the full functionality of Sicstus Prolog FD constraint solver. The same approach can be straightforwardly applied to other domains. In a multi-domain setting, although only one domain can be used at the same time, i.e., there is no hybrid computation for different domains as presented for instance in [FH99], several domains could be managed from the same $\mathcal{TOY}$ program, therefore allowing to solve different-by-nature subproblems on a compositional approach.

A possible drawback that may be seen in our approach is that a programmer not only has to know the $\mathcal{TOY}$ syntax, but also the Prolog syntax. However, the

functional logic approach is understood as the combination of both functional and logic paradigms, which have to be familiar to the programmer. In this line, Prolog syntax is also allowed in $\mathcal{TOY}$ for newcomers from the logic arena. In order to 'overcome' this possible drawback, our aim in improving semantics of the $\mathcal{TOY}$ language is to integrate FD constraints in a uniform framework, in the spirit that real constraints are already integrated [AH+96]. This will imply the adjustment of both the semantics and operational model. As an obvious advantage, it will allow to maintain the $\mathcal{TOY}$ syntax in defining domain variables, posting constraints, and defining the search procedure.

Another important point is to bring the outside world to $\mathcal{TOY}$, by allowing the connection with persistent data, i.e., databases. This is an already identified topic in commercial constraint systems, as for example, ILOG Solver [Pug94] for constraint programming and AMPL [FGK93] for mathematical programming. [Fou97] contains some interesting material about database structures for mathematical programming models, which can be straightforwardly applied to constraint programming models.

The need of a more declarative programming language for solving FD problems has been felt in the research community, which leads to algebraic specification languages [Hen99,Fou01]. Some work remains to be done in topics as modularity, abstraction and combination of ideas coming from other paradigms. For instance, object oriented programming can be thought as a way for building program components defining constraint skeletons, which may model generic objects, that define constraint structures which can be inherited in order to build more specific objects. Moreover, when the domain of the application is more specific, one can think of a specific programming language with built-in constructors for modeling well-known components of the problem domain (like pipes that have to obey certain constraints, nodes that must conform a flow balance, etc.) The combination of experience from different fields of programming languages can be profited to achieve more declarative and useful language for real-problem solving.

References

[AA+01] M. Abengózar, P. Arenas, J.C. González, R. Caballero, A. Gil, J. Leach, F.J. López, N. Martí, M. Rodríguez, J.J. Ruz, and J. Sánchez. $\mathcal{TOY}$. *A Multiparadigm Declarative Language*, Technical Report, 2001. The system is available at `http://titan.sip.ucm.es/toy`

[AGL94] P. Arenas-Sánchez, A. Gil-Luezas and F.J. López-Fraguas. *Combining Lazy Narrowing with Disequality Constraints*, In Proceedings of the International Symposium on Programming Language Implementation and Logic Programming (PLILP'94), Springer LNCS 844, pp. 385–399, 1994

[AH+96] P. Arenas-Sánchez, T. Hortalá-González, F.J. López-Fraguas and E. Ullán-Hernández. Functional Logic Programming with Real Numbers. In *Proceedings of the JICSLP'96 Post-Conference Workshop on Multi-Paradigm Logic Programming.* TR 96-28, Technical University Berlin, 1996.

[Ant92] S. Antoy. *Definitional Trees*. In Proceedings of the International Conference on Algebraic and Logic Programming (ALP'92), Springer LNCS 528, pp. 371–382, 1992.

[BF+00] R.E. Bixby, M. Fenelon, Z. Gu, E. Rothberg and R. Wunderling. MIP: Theory and Practice Closing the Gap. System Modeling and Optimization: Methods, Theory and Applications, Kluwer, The Netherlands, M. J. D. Powell and S. Scholtes, editors, pp. 19–49, 2000.
`http://www.ilog.com/products/optimization/tech/researchpapers.cfm#C++`

[DL+97] K. Darby-Dowman, J. Little, G. Mitra and M. Zaffalon. Constraint Logic Programming and Integer Programming Approaches and Their Collaboration in Solving an Assignment Scheduling Problem. *Constraints* 1, pp. 245–264, 1997.

[Fer00] A.J. Fernández. Implementation of CLP(L). Technical Report. University of Málaga, Spain. 2000.
`http://www.lcc.uma.es/~afdez/generic/`

[FGK93] R. Fourer, D.M. Gay and B.W. Kernighan. AMPL: A Modeling Language for Mathematical Programming, Scientific Press, 1993.

[FH99] A.J. Fernández and P.M. Hill. An Interval Lattice-based Constraint Solving Framework for Lattices. In *Aart Middeldorp and Taisuke Sato, editors, 4th International Symposium on Functional and Logic Programming (FLOPS'99)*, Springer Verlag, LNCS 1722, pp. 194–208, Tsukuba, Japan, November 1999.

[Fou97] R. Fourer. Database Structures for Mathematical Programming Models. Decision Support Systems 20, pp. 317-344, 1997.

[Fou99] R. Fourer. Software Survey: Linear Programming. OR/MS Today 26:4, 64–65, August 1999.
`http://lionhrtpub.com/orms/orms-8-99/survey.html`

[Fou01] R. Fourer and D.M. Gay. Extending an Algebraic Modeling Language to Support Constraint Programming. In Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'01), United Kingdom, 2001.

[GHR97] J.C. González-Moreno, T. Hortalá-González *A Higher Order Rewriting Logic for Functional Logic Programming*. In Proceedings of ICLP'97, The MIT Press, pp. 153–167, 1997.

[GH+99] J.C. González-Moreno, T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. Journal of Logic Programming, Vol. 40, N. 1, pp. 47–87, July, 1999.

[Gon93] J.C. González-Moreno. *A Correctness Proof for Warren's HO into FO Translation*. In Proceedings of GULP'93, pp. 569–585, 1993.

[Han94] M. Hanus. *The Integration of Functions into Logic Programming: A Survey*. Journal of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", pp. 583–628, 1994.

[Han00] M. Hanus. Curry: An Integrated Functional Logic Language, 2000.
`http://www.informatik.uni-kiel.de/~curry/`.

[HAS97] *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language*. Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.

[Hen99] P. Van Hentenryck. The OPL Optimization Programming Language. The MIT Press. 1999.

[JD95] C. Jordan and A. Drexl. A Comparison of Constraint and Mixed-Integer Programming Solvers for Batch Sequencing with Sequence-Dependent Setups. *ORSA Journal on Computing* 7, pp. 160–165, 1995.

[JM94] J. Jaffar and M.J. Maher. *Constraint Logic Programming: A Survey*. Journal of Logic Programming 19/20, pp. 503-582, 1994.

[KLMR92] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro and M. Rodríguez-Artalejo. *Implementing a Lazy Functional Logic Language with Disequality Constraints*, In Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'92), The MIT Press, pp. 207–221, 1992.

[LLR93] R. Loogen, F.J. López-Fraguas and M. Rodríguez-Artalejo. *A Demand Driven Computation Strategy for Lazy Narrowing*. In Proceedings of PLILP'93, Springer LNCS 714, 184–200, 1993.

[Lop92] F.J. López-Fraguas. A General Scheme for Constraint Functional Logic Programming. Proceedings of the International Conference on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, pp. 213–227, 1992.

[LS99] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99), pp. 244-247. Springer LNCS 1631, 1999.
The system is available at `http://titan.sip.ucm.es/toy`

[Lux01] W. Lux. Adding Linear Constraints over Real Numbers to Curry. FLOPS 2001, LNCS 2024, pp. 185-200, 2001.

[Pug94] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems)*, 1994.
See also `http://www.ilog.com`

[Rod01] M. Rodríguez-Artalejo. Functional and Constraint Logic Programming. H. Comon, C. Marché and R. Trainen, editors, Constraints in Computational Logics, Springer LNCS 2002, pp. 202-270, 2001.

[Sic99] *SICStus Prolog User's Manual*, Intelligent Systems Laboratory, Swedish Institute of Computer Science, Release 3.8, October 1999.

[SL98] J. Sánchez-Hernández and F. López-Fraguas. $\mathcal{TOY}$: Un lenguaje lógico funcional con restricciones. Research Report. DSIP UCM. 1998.

[Smi95] B.M. Smith. A Tutorial on Constraint Programming, Report 65.14, University of Leeds, April 1995.
`ftp://agora.leeds.ac.uk/scs/doc/reports/1995/95_14.ps.Z`

[Smi96] B.M. Smith, S.C. Brailsford, P.M. Hubbard and H.P. Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. Constraints. An International Journal 1(1/2), pp. 119-138, September 1996.

[Smo95] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Current Trends in Computer Science*. Springer LNCS 1000, 1995.

[SSW99] B. Smith, K. Stergiou and T. Walsh. Modeling the Golomb Ruler Problem, Research Report 99.12, Department of Computer Science, University of Strathclyde, Glasgow, Scotland, UK, January 1999.

[Tsa93] E. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.