Solving FD Constraints in $\mathcal{TOY}(\mathcal{FD})$

Antonio J. Fernández[1], Teresa Hortalá-González[2], and Fernando Sáenz-Pérez[2]

[1] Dpto. Lenguajes y Ciencias de la Computación, Univ. de Málaga, Spain***
[2] Dpto. Sist. Inf. y Prog. Univ. Complutense de Madrid, Spain
afdez@lcc.uma.es, {teresa,fernan}@sip.ucm.es

Abstract. This paper highlights the power of $\mathcal{TOY}(\mathcal{FD})$, a functional logic language with support for finite domain constraints, and shows, by means of examples, how combinatorial and optimization problems are easily coded and solved in $\mathcal{TOY}(\mathcal{FD})$. Moreover, a comparison with respect to the traditional CLP(FD) approach demonstrates that $\mathcal{TOY}(\mathcal{FD})$ is not only an alternative but is more flexible and provides higher expressivity.
The paper also introduces a novel proposal in functional logic languages to recover and manage, at the user level, internal information about the constraint solving processing at runtime. This proposal increases the constraint capacity of the language as it allows the user to implement specific constraint mechanisms (e.g., new search strategies).
Keywords: Functional Logic Programming Languages, Constraint Logic Programming, Rewriting Systems, Constraint Solvers.

1 Introduction

In [3] we proposed the integration of finite domain (FD) constraints into the functional logic programming (FLP) language $\mathcal{TOY}$ [9, 11] and, as a result, we presented the language $\mathcal{TOY}(\mathcal{FD})$ that integrates the best features of existing functional and logic languages, as well as FD constraint solving. We described the basics about syntax and type discipline of $\mathcal{TOY}(\mathcal{FD})$ programs. We also presented an implementation of the language and showed that it keeps a similar efficiency to existing constraint logic programming over finite domains (CLP(FD)) solvers (e.g., SICStus Prolog) and better performance than a related FLP(FD) system (PAKCS). Afterwards, in [4] we provided a sketch of the operational semantics consisting of a novel combination of lazy evaluation and FD constraint solving not existing, to our knowledge, in any published constraint solver.

Now, this paper illustrates, by means of examples, the features of $\mathcal{TOY}(\mathcal{FD})$, and shows its flexibility to solve combinatorial (optimization) problems. Moreover, the paper demonstrates (again by examples) that $\mathcal{TOY}(\mathcal{FD})$ is more flexible and expressive than the existing CLP(FD) approaches.

The paper also presents a glass box mechanism provided in $\mathcal{TOY}(\mathcal{FD})$ that is a contribution to functional logic programming as it enables the user to recover,

*** Fernández was partially supported by the projects TIC2001-2705-C03-02 and TIC2002-04498-C05-02 funded by the Spanish Ministry of Science and Technology.

at run-time, internal information about the constraint solving. This mechanism consists of a set of predefined functions, called *reflection functions*, that can be used, for instance, to define new search methods at the user level.

2 An Overview of $\mathcal{TOY}(\mathcal{FD})$

$\mathcal{TOY}(\mathcal{FD})$ programs are $\mathcal{TOY}$ programs where FD constraints are defined as functions which are solved by an efficient solver connected to $\mathcal{TOY}$ (see [4]). $\mathcal{TOY}(\mathcal{FD})$ programs consist of *datatypes*, *type alias*, *infix operator* definitions, and rules (see below) for defining *functions*. The syntax is mostly borrowed from Haskell with the remarkable exception that variables begin with upper-case whereas constructor symbols use lower-case, as function symbols do. In particular, functions are *curried* and the usual conventions about application associativity hold.

Each function f has an associated principal type of the form $\tau_1 \to \ldots \to \tau_m \to \tau$ (where τ does not contain $\to$). As usual in functional programming (FP), types are inferred and, optionally, can be declared in the program. Defining rules for a function f have the basic form $f\ t_1\ \ldots\ t_n = r \Leftarrow C$. Informally, the intended meaning of a program rule is that a call to f can be reduced to r whenever the actual parameters match the patterns t_i, and the conditions in C are satisfied. Predicates are viewed as a particular kind of functions, with type $p :: \tau_1 \to \ldots \to \tau_n \to$ `bool`. As a syntactic facility, we can use *clauses* as a shorthand for defining rules whose right-hand side is *true*. This allows to write Prolog-like predicate definitions; each clause $p\ t_1 \ldots t_n :-\ C_1, \ldots, C_k$ abbreviates a defining rule of the form $p\ t_1 \ldots t_n = true \Leftarrow C_1, \ldots, C_k$.

2.1 FD Constraints in $\mathcal{TOY}(\mathcal{FD})$

An *FD constraint* in $\mathcal{TOY}(\mathcal{FD})$ is a predefined function. Table 1 shows a small subset of the FD constraints supported by $\mathcal{TOY}(\mathcal{FD})$, where `int`, `[`$\tau$`]` and `labelType` are predefined types, respectively, for integers, the type 'list of τ' and a type used to define search strategies for finite domain labeling [5].

Table 1. Some Predefined FD Constraints and Enumeration Functions

RELATIONAL CONSTRAINT OPERATORS	ARITHMETICAL CONSTRAINT OPERATORS
#>, #<, #>=, #<=, # =, #\= :: int → int → bool	#*, #/, #+, #− :: int → int → int
COMBINATORIAL CONSTRAINTS	MEMBERSHIP CONSTRAINTS
all_different :: [int] → bool	domain :: [int] → int → int → bool
PROPOSITIONAL CONSTRAINTS	ENUMERATION FUNCTIONS
#<=> :: bool → bool → bool	labeling :: [labelType]→[int]→ bool

$\mathcal{TOY}(\mathcal{FD})$ supports *relational constraint operators* including equality, disequality, *arithmetical constraint operators*, a wide set of well-known *global constraints* (e.g., `all_different`/1, which ensures different values for the elements

in its list argument), *membership constraints* (e.g., `domain`/3, to restrict a list of variables to have values in an interval of integers), *propositional constraints* (e.g., #<=>/2, to define constraint reification), and *enumeration functions* (e.g., `labeling`/2, with a number of strategies) with optimization. As in other constraint languages, we explicitly distinguish the relational operators which are overloaded as constraints (in our case, with the symbol #).

For reasons of space, we do neither mention nor explain all the constraints in detail and encourage the interested reader to visit the link proposed in [5] for a more detailed explanation. We emphasize that all the pieces of code in this paper are executable in $\mathcal{TOY}$(FD) and the answers for example goals correspond to actual executions of the programs.

3 $\mathcal{TOY}(\mathcal{FD})$ vs. {CLP(FD) , F(L)P}

We discuss the advantages of $\mathcal{TOY}(\mathcal{FD})$ with respect to classical CLP(FD) , FP and FLP languages. When necessary, we illustrate different features of $\mathcal{TOY}(\mathcal{FD})$ by means of examples.

It is well-known that CLP(FD) is a successful declarative instance of constraint programming. For this reason, it is obliged to make note the advantages of $\mathcal{TOY}(\mathcal{FD})$ with respect to CLP(FD). This section explains why the presence of functions in terms of FD constraint handling provides additional advantages to those provided in the logic programming setting. Also, $\mathcal{TOY}(\mathcal{FD})$ increases the power of F(L)P with FD constraint solving, which means to extend the range of problems over which the functional approach can be profitably applied, and adds efficiency in the solving of combinatorial problems.

CLP(FD) $\subset$ $\mathcal{TOY}(\mathcal{FD})$. In general, CLP(FD) languages may be considered as a subset of $\mathcal{TOY}(\mathcal{FD})$. The reason is that, as $\mathcal{TOY}(\mathcal{FD})$ keeps a logic component (and includes Prolog-like predicate definitions), any CLP(FD) program can be straightforwardly translated into a $\mathcal{TOY}(\mathcal{FD})$ program. A direct consequence is that our language is able to cope with a wide range of applications. We will not insist here on this matter, but prefer to concentrate on the extra capabilities of $\mathcal{TOY}(\mathcal{FD})$ with respect to CLP(FD).

Extra Capabilities wrt. CLP(FD). Due to its functional component, $\mathcal{TOY}(\mathcal{FD})$ adds further expressiveness to CLP(FD) as it allows the declaration of functions and their evaluation in the FP style. Additionally, it provides a collection of useful functions (e.g., `map, iterate, filter`) usually presented in functional languages. In the following, we enumerate and discuss other features not presented (or unusual) in the CLP(FD) paradigm.

Types. Our language is strongly typed and thus involves all the well-known advantages of a type checking process. FD constraints are functions with clear declarations, which means their wrong uses can be straightforwardly detected in the typical type checking process. It is commonly acknowledged that types enhance program development and maintenance.

Functional Notation. It is well-known that functional notation reduces the number of variables with respect to relational notation, and thus, $\mathcal{TOY}(\mathcal{FD})$ increases the expressiveness of CLP(FD) as it combines relational and functional notation. For instance, in CLP(FD) the constraint conjunction `N=2, X ∈ [1,10-N]` cannot be expressed directly and must be written as either `N=2, Max is 10-N, domain([X],1,Max)` or `N=2, Max is 10-N, X in 1..Max` that uses an extra variable (and just for a very simple constraint). However, $\mathcal{TOY}(\mathcal{FD})$ expresses that constraint directly as `N==2, domain [X] 1 (10-N)`.

Currying. Again, due to its functional component, $\mathcal{TOY}(\mathcal{FD})$ allows the currying of functions (and thus constraints); for instance, see in Example 1 below the application of curried FD constraint `(3 #<)/1`.

Higher Order and Polymorphism. In $\mathcal{TOY}(\mathcal{FD})$, functions are first-class citizens, i.e., a function (and thus an FD constraint) can appear in any place where a data can. As a direct consequence, an FD constraint may appear as an argument (or even as a result) of another function or constraint. The functions managing other functions are called higher order (HO) functions.

Example 1. A traditional example of a polymorphic HO function is the function map:

```
map :: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [(F X) | (map F Xs)]
```

where lists adhere to the syntax of Prolog lists. Now, suppose that X and Y are FD variables ranging in the domain [0,100] (e.g., via `domain [X,Y] 0 100`). Then, the goal `map (3#<) [X,Y]` returns the Boolean list `[true,true]` resulting from evaluating the list `[3#<X,3#<Y]`, and `X` and `Y` are also restricted to have values in the range `[4,100]` as the constraints `3#<X` and `3#<Y` are sent to the constraint solver. Note also the use of the curried function `(3#<)`.

Laziness. In logic languages, the arguments (of predicates) are evaluated before the call (*call-by-value*). This is known as *eager evaluation* (all the possible evaluations are performed). An alternative to this evaluation supported by the functional setting is the *call-by-need*, in which arguments are evaluated to the required extent. In a functional setting, *call-by-need* evaluation often corresponds to *lazy evaluation* whereas *call-by-value* often corresponds to *eager evaluation*[1]. $\mathcal{TOY}(\mathcal{FD})$ increases the power of CLP(FD) by incorporating a novel mechanism (to our knowledge, presented in no CLP(FD) language) that combines *lazy evaluation* and FD constraint solving, in such a way that only the really necessary constraints are sent to the solver. In general, lazy narrowing avoids computations which are not demanded, therefore saving computation time. This opens new possibilities for FD constraint solving (e.g., it allows to manage infinite structures).

[1] Strictly speaking, lazy evaluation may also correspond to the notion of *only once evaluated* in addition to *only required extent*.

Example 2. Consider the following recursive function that generates an infinite list of FD variables ranging in the interval [0,N-1] for some N passed as argument.

```
generateFD :: int -> [int]
generateFD N =[ X | generateFD N ] <== domain [X] 0 (N-1)
```

and the goal `take 3 (generateFD 10) == List`, where `take N L` returns a list with the first `N` elements of `L`. An eager evaluation of this goal does not terminate as it tries to completely evaluate the second argument, yielding to an infinite computation. However, a lazy evaluation generates just the first 3 elements of the list (i.e., a list with 3 FD variables ranging in the interval [0,9]).

4 Solving Constraint Problems

In the following, we show two practical constraint satisfaction problems that can be easily coded in $\mathcal{TOY}(\mathcal{FD})$. The first one is an optimization problem.

4.1 Golomb Rulers

Golomb Rulers are a class of undirected graphs that, unlike usual rulers, measure more discrete lengths than the number of marks it carries. Their particularity is that on any given ruler, all differences between pairs of marks are unique. This feature makes Golomb Rulers to be really interesting for practical applications such as radio astronomy, X-ray crystallography, circuit layout, geographical mapping, radio communications, and coding theory.

Traditionally, researchers are usually interested in discovering rulers with minimum length and Golomb rulers are not an exception. An *Optimal Golomb Ruler* (OGR) is defined as the shortest Golomb ruler for a number of marks. OGRs may be multiple for a specific number of marks. However, the search for OGRs is a task extremely difficult as this is a combinatorial problem whose bounds grow geometrically with respect to the solution size [12]. This has been a major limitation as each new ruler to be discovered is by necessity larger than its predecessor. Fortunately, the search space is bounded and, therefore, solvable [8]. To date, the highest Golomb ruler whose shortest length is known is the ruler with 23 marks [13]. Solutions to OGRs with a number of marks between 10 and 19 were obtained by very specialized techniques, and best solutions for OGRs between 20 and 23 marks were obtained by massive parallelism projects (these solutions took several months to be found) [13].

$\mathcal{TOY}(\mathcal{FD})$ enables the solving of optimization problems by using the function `labeling` with the value `toMinimize X` and/or `toMaximize X` (these values are intended for the minimization and maximization, respectively, of an FD variable `X`). Below, we show a $\mathcal{TOY}(\mathcal{FD})$ program to solve OGRs with N marks and the solving of a goal for `N = 12`.

```
golomb :: int -> [int] -> bool
golomb N L = true <== length L == N, NN == trunc(2^(N-1)) - 1,
```

```
     domain L 0 NN, append [0|_] [Xn] == L, % Typical Append
     distances L Diffs, domain Diffs 1 NN,
     all_different Diffs, append [D1|_] [Dn] == Diffs,
     D1 #< Dn, labeling [toMinimize Xn] L   % Optimization Search

distances :: [int] -> [int] -> bool
distances   [ ]  [ ] = true
distances [X|Ys]  D0 = true <==
     distancesB X  Ys D0 D1, distances Ys D1

distancesB :: int -> [int] -> [int] -> [int] -> bool
distancesB _ [ ] D D = true
distancesB X [Y|Ys] [Diff|D1] D0 = true <==
     Diff #= Y#-X, distancesB X Ys D1 D0
Toy(FD)> golomb 12 L
         yes  L ==  [0,2,6,24,29,40,43,55,68,75,76,85]
```

$\mathcal{TOY}(\mathcal{FD})$ solves 10-marks OGRs in 17 seconds and 12-marks OGRs in 10,918 seconds (i.e., about three hours), in a Pentium 1.4 Ghz under Windows. See [3] for performance results.

4.2 DNA Sequencing

In this section, we show a simplified version of restriction site mapping (RSM) taken from [7]. A DNA sequence is a finite string over the elements {A,C,G,T}. An enzyme partitions a DNA sequence into certain fragments. The problem consists of reconstructing the original DNA sequence from the fragments and other information taken from experiments. To keep the problem concise, we consider a simplification of this problem, which only deals with the length of the fragments, instead of the fragments themselves.

Consider the use of two enzymes. The first enzyme partitions the DNA sequence into $\mathtt{A}_1,\ldots,\mathtt{A}_N$ and the second into $\mathtt{B}_1,\ldots,\mathtt{B}_M$. A simultaneous use of the two enzymes also produces a partition into $\mathtt{D}_1,\ldots,\mathtt{D}_K$, which corresponds to the combination of the previous two partitions, that is:

$\forall \mathtt{i}\ \exists \mathtt{j}\colon\ \mathtt{A}_1\ldots\mathtt{A}_i = \mathtt{D}_1\ldots\mathtt{D}_j \wedge \forall \mathtt{i}\ \exists \mathtt{j}\colon\ \mathtt{B}_1\ldots\mathtt{B}_i = \mathtt{D}_1\ldots\mathtt{D}_j$,
and, conversely, $\forall \mathtt{j}\ \exists \mathtt{i}\colon\ \mathtt{D}_1\ldots\mathtt{D}_j = \mathtt{A}_1\ldots\mathtt{A}_i \vee \mathtt{D}_1\ldots\mathtt{D}_j = \mathtt{B}_1\ldots\mathtt{B}_i$,
where '$\mathtt{A}_1\ldots\mathtt{A}_i$' denotes the sequence of fragments $\mathtt{A}_1$ to $\mathtt{A}_i$, and '=' denotes syntactic equality.

Let $\mathtt{a}_i$ ($\mathtt{b}_i$ and $\mathtt{d}_i$, resp.) denote the length of $\mathtt{A}_i$ ($\mathtt{B}_i$ and $\mathtt{D}_i$, resp.). Let $\mathbf{a_i}$ denote the subsequence $a_1 \ldots a_i$, $\mathtt{1} \leq \mathtt{i} \leq \mathtt{N}$ (and similarly for $\mathbf{b_i}$ and $\mathbf{d_i}$)

The problem is stated as follows: given the multisets $a = \{\mathtt{a}_1,\ldots,\mathtt{a}_N\}$, $b = \{\mathtt{b}_1,\ldots,\mathtt{b}_N\}$, and $d = \{\mathtt{d}_1,\ldots,\mathtt{d}_N\}$, construct the sequences $\mathbf{a}_N = \mathtt{a}_1\ldots\mathtt{a}_N$, $\mathbf{b}_M = \mathtt{b}_1\ldots\mathtt{b}_M$, and $\mathbf{d}_K = \mathtt{d}_1\ldots\mathtt{d}_K$.

The algorithm to solve this problem generates $\mathtt{d}_1$, $\mathtt{d}_2$, ... in order and extends the partitions for **a** and **b** using the following invariant property which can be obtained from the problem definition above. Either

– $\mathtt{d}_k$ is aligned with $\mathtt{a}_i$, that is, $\mathtt{d}_1 + \cdots + \mathtt{d}_k = \mathtt{a}_1 + \cdots + \mathtt{a}_i$, or
– $\mathtt{d}_k$ is aligned with $\mathtt{b}_j$, but not with $\mathtt{a}_i$, (for simplicity, we assume we never have all three partitions aligned except at the beginning and at the end), that is, $\mathtt{d}_1 + \cdots + \mathtt{d}_k = \mathtt{a}_1 + \cdots + \mathtt{a}_i$.

The following Boolean function `solve/6` takes three input lists representing a, b, and d, in its three first arguments respectively. The output represents the possibilities to construct d from the fragments taken from a, and b.

```
solve:: [int] -> [int] -> [int] -> [int] -> [int] -> [int] -> bool
solve A B D [AF|MA] [BF|MB] [DF|MD] :-
      choose_initial A B D AF BF DF A2 B2 D2,
      rsm A2 B2 D2 AF BF DF MA MB MD, labeling [] [BF|MB]

rsm :: [int] -> [int] -> [int] -> int -> int ->
       int -> [int] -> [int] -> [int] -> bool
rsm [] [] [] LenA LenB LenD [] [] [] = true

rsm A B D LenA LenB LenA [Ai|MA] MB [Dk|MD] :-  LenA #< LenB,
      Dk #<= LenB #- LenA, Ai #>= Dk, choose Ai A == A2,
      choose Dk D == D2, NLenA #= LenA #+ Ai, NLenD #= LenA #+ Dk,
      rsm A2 B D2 NLenA LenB NLenD MA MB MD

rsm A B D LenA LenB LenB MA [Bj|MB] [Dk|MD] :-  LenB #< LenA,
      Dk #<= LenA #- LenB, Bj #>= Dk, choose Dk D == D2,
      choose Bj B == B2, NLenB #= LenB #+ Bj, NLenD #= LenB #+ Dk,
      rsm A B2 D2 LenA NLenB NLenD MA MB MD

choose_initial :: [int] -> [int] -> [int] -> int -> int ->
                  int -> [int] -> [int] -> [int] -> bool
choose_initial A B D AF BF DF A2 B2 D2 :-
      choose AF A == A2, choose BF B == B2, choose DF D == D2

choose :: int -> [int] -> [int]
choose X [] = []
choose Ai [Ai|A2] = A2
choose Ai [A1, A2|A] = [A1|choose Ai [A2|A]]
```

For instance, one goal for this program could be:

```
TOY(FD)> solve [3,2,4,5,9] [7,8] [3,2,3,4,2,2,3,4] L1 L2 L3
      yes   L1 ==  [ 4, 2, 5, 9, 3 ]
            L2 ==  [ 8, 7, 3, 5 ]
            L3 ==  [ 4, 2, 2, 3, 4, 3, 2, 3 ]
```

which means that `L1` (`L2` resp.) constructs `L3` by aligning the fragments as the following table indicates:

L1	L3	L2	L3
4	4	8	4,2,2
2	2	7	3,4
5	2,3	3	3
9	4,3,2	5	2,3
3	3		

In the program code, `rsm/9` provides the choice of partitioning with either one of the two available enzymes. The last three arguments hold the length of the subsequences found so far. The function `choose_initial/9` chooses the first fragment and the first call to `rsm` is made with this invariant holding. Finally, the procedure `choose/2` deletes some element from the given list and returns the resultant list.

Note that the Boolean functions `solve/6`, `rsm/9`, and `choose_initial/9` have been written in a Prolog-like fashion, thanks to the syntactic sugaring allowed in our system, whereas `choose/2` has been written as a function which returns the list resulting from deleting an element of its input list.

5 Lazy Applications

This section highlights the expressive power of $\mathcal{TOY}(\mathcal{FD})$ by proposing very expressive solutions to a number of constraint satisfaction problems that can be described and solved via lazy evaluation of infinite lists. This level of expressiveness and conciseness cannot be directly reached by using a CLP(FD) language.

In general, all the examples described here show, among other features (e.g., constraint composition or curried notation) the combination of higher order (HO) applications and (indeterministic) lazy generation of constraints.

5.1 Process Network: Client-Server Interaction

Processes can be considered as functions consuming data (i.e., arguments) and producing values for other functions. Processes are often suspended until the evaluation of certain expression is required (by other process). In these cases, lazy evaluation corresponds to particular coroutines for the processes.

One interesting application is to solve the communication between a client and a server with the Input/Output model via *Streams*: If the client generates requests from one initial requirement, the server will generate answers that will be again processed by the client and so on. For simplicity, we consider that requests and answers are integer numbers. This process network can be clearly defined in $\mathcal{TOY}(\mathcal{FD})$ with recursive definitions as follows:

```
requests, answers :: [int]
requests = client initial answers
answers = server requests
```

Suppose now that the client returns the request and generates a new one (i.e., a next one) from the first answer of the server and that the server processes each request to generate a new answer. This is defined in $\mathcal{TOY}(\mathcal{FD})$ as follows:

```
client :: int -> [int] -> [int]
client Ini [R|Rs] = [Ini | client (next R) Rs]
server :: [int] -> [int]
server [P|Ps] = [process P | server Ps]
```

The architecture is completed by defining adequately the initial requirement, the processing function and the selection of the next request. As example, and for simplicity, we can define them as follows:

```
process :: int -> int   initial :: int     next :: int -> int
process = (+3)          initial = 4        next = id  %Idempotence
```

Note that this is not enough to produce an outcome as the goal `requests` goes into a non-ending loop. However, the lazy evaluation mechanism of $\mathcal{TOY}(\mathcal{FD})$ allows to evaluate a finite number (`N`) of requests; this can be done by redefining the functions `client, answers` and `requests` as follows:

```
client :: int -> [int] -> [int]
client Ini Rs = [Ini | client (next (head Rs)) (tail Rs)]
answers :: int -> [int]
answers N = server (requests N)
requests :: int -> [int]
requests N = take N (client initial (take N (answers N)))
```

where `head/1` and `tail/1` return the head and tail of a list respectively. Below, we show an example of solving that evaluates exactly the first 15 requests.

```
TOY(FD)> requests 15 == L
         yes L == [4,7,10,13,16,19,22,25,28,31,34,37,40,43,46]
```

Observe that this example illustrates no constraint feature of $\mathcal{TOY}(\mathcal{FD})$ but it has been described to show the flexibility of the language and introduce a more interesting example in the next section that makes use of the constraint facilities of the language.

5.2 Pipelines

Pipelines can be a powerful tool to solve heterogeneous constraint satisfaction problems, and these are easily expressed at a high level in $\mathcal{TOY}$ (FD) via applying HO constraints and curried notation. For example, consider the Optimal Golomb ruler (OGR), N-queens and client-server programs shown in preceding sections. Then, the goal (for some natural N)

```
map (map (queens [ff]))(map golomb (requests N)) == L
```

corresponds directly to the scheme shown in Figure 1 if we redefine the function *process* of Section 5.1 as `process = (+1)`. The solving of this goal, as in the preceding example of client-server architecture, generates N answers in the form of a N-elements list `A` from an initial request `initial = 4`; each element $\mathtt{a}_i \in$ `A` (i.e., each answer of the server with `i` $\in \{$`1,...,N`$\}$ and $\mathtt{a}_i$ `= initial + i`

- 1) is used to feed the OGR solver with a_i marks producing a new list S = [s_{a_1},...,s_{a_N}] containing N solutions for OGRs with marks a_1,...,a_N. Finally, each element o_k (with k∈{1,...,a_i}) belonging to the solution to the OGR with a_i marks in S (i.e., s_{a_i} = [o_1,...,o_{a_i}]) feeds the o_k-queens solver and the first solution to the o_k-queens problem is computed.

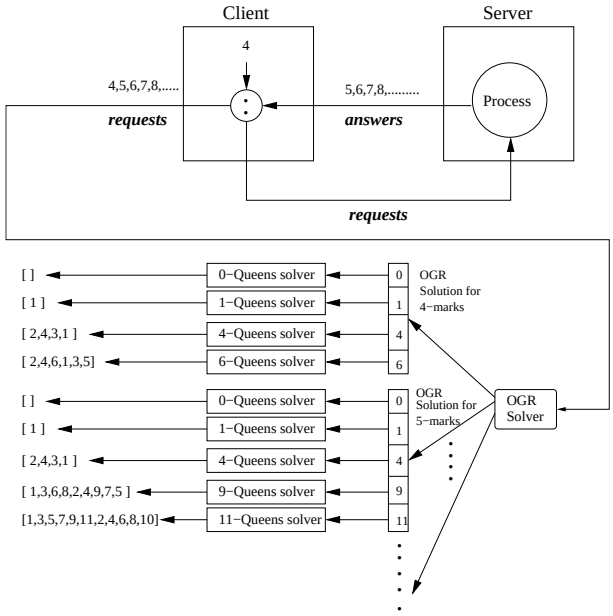

Fig. 1. Pipeline with Client-Server Architecture

For example, the goal shown above (for N = 2) first calculates the solutions for the OGR with 4 marks (i.e., [0,1,4,6]) and 5 marks (i.e., [0,1,4,9,11]), and feeds the queens solver with each mark returning the first solution for 0,1,4,6,0,1,4,9 and 11 queens.

```
TOY(FD)> map (map (queens [ff])) (map golomb (requests 2)) == L
  yes
  L==[[ [], [ 1 ], [ 2, 4, 1, 3 ], [ 2, 4, 6, 1, 3, 5 ] ],
      [ [], [ 1 ], [ 2, 4, 1, 3 ], [ 1, 3, 6, 8, 2, 4, 9, 7, 5 ],
            [ 1, 3, 5, 7, 9, 11, 2, 4, 6, 8, 10 ]]]
```

This example illustrates how easy and natural may be the combination of different problems in $\mathcal{TOY}(\mathcal{FD})$ without adding extra code.

6 Reflection Functions

Here we present an innovative mechanism in FLP languages to allow users to define their own specialized constraint constructs.

6.1 Black Box vs. Glass Box

In the beginning, CLP systems provided particular built-in constraints to solve specific applications. These constraints are black boxes from the user point of view in the sense that the user can make use of them but does not need to understand in full detail their execution behavior. So far, we have shown that $\mathcal{TOY}(\mathcal{FD})$ also follows a black box approach. The advantage is evident: the associated FD constraint solver of the system uses specialized propagation mechanisms for the FD constraints leading thus to a major efficiency in the solving of well-known and complex problems (i.e., specific problems). However, there are also clear disadvantages: firstly, they are built-in into the system and coded internally in very complex manners. Thus, it is very difficult to understand their operational behavior. Secondly, these complex constraints lack adaptability for being used in non-standard problems that require, for instance, specific search strategies or unusual constraints.

To overcome this lack of flexibility and applicability of black box constraints, some existing constraint systems follow an alternative approach called *glass-box*. This approach allows the user to define new constraints in terms of primitive constraints provided by the system. The advantages are clear. In practice, there are many constraints that are not specific to standardized applications and the users can define their own *glass box* constraints specialized for particular applications. Also, *glass box* constraints are useful in the re-utilization of code since they are easily adapted to solve similar problems to those problems for which they were designed. Moreover, the user understands totally the operational behavior of their constraints so that it is very easy to modify them in order to find for the most adequate constraint solving for a problem. This global understanding of the process allows for general optimizations, as opposed to the many local and particular optimizations hidden inside the black box constraints. Two main criticisms to the glass box approach can be done. Firstly, as glass box constraints are not specific to particular applications, their efficiency depends directly on how the user defines them. Secondly, the user has to be cautious about correctness and completeness of the definition of the glass box constraints.

As both approaches have advantages and disadvantages, it seems clear that a system combining both approaches should be desirable, and this is the current proposal of $\mathcal{TOY}(\mathcal{FD})$. To our knowledge, this is the first time that a pure constraint FLP language provides this capability.

6.2 Recovering Internal Information at Runtime

The glass box approach of $\mathcal{TOY}(\mathcal{FD})$ is based on a set of predefined functions called *reflection functions* supported by the system, that allow, at runtime, to recover internal information about the constraint solving process. These functions increase the flexibility of the language as they allow the user to construct specific constraint mechanisms such as new search strategies. Below, we show part of this set of reflection functions:

```
fd_min, fd_max, fd_size, fd_degree :: int -> int
```

```
fd_neighbors :: int -> [int]
fd_var :: int -> bool
empty_interval :: int -> int -> bool
empty_fdset :: [fdset] -> bool
fd_set :: int -> [fdset] -> bool
fdset_size  :: [fdset] -> int
```

where `fd_set` is a built-in type that captures the internal representation of a domain. There are constraints that recover information about the constrained variables. For instance, `fd_min X` and `fd_max X` return, respectively, the minimum and maximum value in the domain associated to `X`; `fd_size X` returns the cardinality of the domain of variable `X`, and `fd_neighbors X` returns the list of variables related, directly or not, with `X` via some constraint. Also `fd_degree X` returns the number of constraints involving variable `X`, and `fd_var X` is true if `X` is an FD variable and its domain is not a singleton value. `empty_fdset` is true if its input FD set is empty.

Other constraints return information specifically about the domains. For instance, `empty_interval Min Max` is true if the interval `[Min,Max]` is empty and `fd_set X Dom` is true is `Dom` unifies with the internal representation of the domain of variable `X`, and `fdset_size Set` calculates the cardinality of the domain represented internally by `Set`. $\mathcal{TOY}(\mathcal{FD})$ provides more reflection functions; see [5] for more details.

6.3 Programmable Search

As an example of practical use of the reflection functions, here we show a user defined $\mathcal{TOY}(\mathcal{FD})$ function `labelff/1` that implements one of the most popular labeling strategies, often supported by the constraint systems, the so-called *first-fail*, that selects the variable with the least number of values in its domain.

```
labelff :: [int] -> bool
labelff [] = true
labelff [X] = false <==  fd_set X SX, empty_fdset SX
labelff [X] = true  <==  domain [X] (fd_min X) (fd_min X)
labelff [X] = true  <==  Next == (fd_min X)+1,
     domain [X] Next (fd_max X), labelff [X]
labelff [X,X1|Xs] = true <==
     choose_min [X,X1|Xs] Y Ys, labelff [Y], labelff Ys

choose_min :: [int] -> int -> [int] -> bool
choose_min [X] X [] = true
choose_min [X,Y|Ys] M [Y|Rs] = choose_min [X|Ys] M Rs <==
     fd_set X SX, fd_set Y SY, fdset_size SX <= fdset_size SY
choose_min [X,Y|Ys] M [X|Rs] = choose_min [Y|Ys] M Rs <==
     fd_set X SX, fd_set Y SY, fdset_size SX > fdset_size SY
```

Observe that when there are several variables to label, this function selects the one with the minimum domain cardinality (via `choose_min/3`) by making use

of information recovered by the reflection functions `fd_set/2` and `fdset_size/1`. Also, when there is just one variable `X`, it reactivates the search process by dividing its domain by the value `(fd_min X)`.

7 Related Work

There exist some attempts to integrate constraints into the functional logic framework. For instance, [9] show how to integrate both linear constraints over real numbers and disequality constraints into the FLP language $\mathcal{TOY}$. Our work is guided to FD constraints, instead of real constraints (although they are preserved), which allows to use non linear constraints and adapts better to a range of combinatorial applications.

$\mathcal{TOY}(\mathcal{FD})$ may also be considered from a multiparadigmatic view, i.e., it combines constraint programming with several paradigms in one setting. In this context, there are some similarities with the language Oz [16] as this provides salient features of FP such as compositional syntax and first-class functions, and features of LP and constraint programming including logic variables, constraints, and programmable search mechanisms. However, Oz is quite different to $\mathcal{TOY}(\mathcal{FD})$ because of a number or reasons: (1) Oz does not provide main features of classical functional languages such as explicit types or curried notation; (2) functional notation is provided in Oz as a syntactic convenience; (3) the Oz computation mechanism is not based on rewriting logic like $\mathcal{TOY}(\mathcal{FD})$; (4) Oz supports a class of lazy functions based on a demand-driven computation, but this is not an inherent feature of the language (as in $\mathcal{TOY}(\mathcal{FD})$) and functions have to be made lazy explicitly (e.g., via the concept of *futures*); (5) functions and constraints are not really integrated, that is to say, they do not have the same category as in $\mathcal{TOY}(\mathcal{FD})$ (i.e., constraints are functions) and both coexist in a concurrent setting, and (6) Oz programs follow a far less concise program syntax than $\mathcal{TOY}(\mathcal{FD})$. In fact *Oz* generalizes the CLP and concurrent constraint programming paradigms to provide a very flexible approach to constraint programming very different to our proposal.

Also, LIFE [1] is an experimental language proposing to integrate logic programming and functional programming but, also, the proposal is quite different to $\mathcal{TOY}(\mathcal{FD})$ as firstly, it is considered in the framework of object-oriented programming, and, secondly, LIFE enables the computation over an order-sorted domain of feature trees by allowing the equality (i.e., unification) and entailment (i.e., matching) constraints over order-sorted feature terms.

There exist other constraint systems that share some aspects with $\mathcal{TOY}(\mathcal{FD})$ although they are very different. One of those systems is FaCiLe [2] an interesting constraint programming library that provides constraint solving over integer finite domains, HO functions, type inference, strong typing, and user-defined constraints. However, despite these similarities, FaCiLe is very different to $\mathcal{TOY}(\mathcal{FD})$ as it is built on top of the functional language OCaml that provides full imperative capabilities and does not have a logical component; also Ocaml is a strict language, as opposed to lazy ones. In fact, as Oz , it allows the

manipulation of potentially infinite data structures by *explicit* delayed expressions, but laziness is not an inherent characteristic of the resolution mechanism. Moreover, FaCiLe is a library and thus it lacks programming language features. A few earlier constraint frameworks were designed over functional languages such as Lisp as the seminal object oriented PECOS system [10] or SCREAMER [14].

Finally, other interesting system is OPL [15] that cannot be compared to our work because it is an algebraic language which, therefore, is not a general programming language.

Generally speaking, $\mathcal{TOY}(\mathcal{FD})$ is, from its nature, different to all the constraint systems discussed above since $\mathcal{TOY}(\mathcal{FD})$ is a *pure* FLP language that combines characteristics of *pure* LP and *pure* FP paradigms, and its operational mechanism is the result of combining the operational methods of logic languages (i.e., unification and resolution) and functional languages (i.e., rewriting).

8 Conclusions

This paper continues our work about the integration of finite domain constraints into a state-of-the-art implementation of a functional logic language by showing its applicability. We have used examples to show the integration of FD constraint solving, lazy evaluation, higher order applications of functions and constraints, polymorphism, composition of functions (and, in particular, constraints), combination of relational and functional notation, and a number of other characteristics. In particular, these features allow to write more concise programs, therefore increasing the expressivity level. Moreover, usual techniques as type checking, make faster program development and maintenance.

We have also shown the advantages of our proposal with respect to others which do not embody LP+FP+FD. We are only aware of a work in this line (PAKCS [6]), although there is no specific publication, to our knowledge, of their results up to date. However, we have compared their limited system with ours in [3], showing that, first, our system clearly outperforms theirs, and, second, we provide a quite larger set of predefined constraints. Also, we have discussed related work and shown that existing multiparadigmatic FD constraint systems are very different from $\mathcal{TOY}(\mathcal{FD})$, whose operational mechanism is based on a novel combination of constraint solving and lazy narrowing.

We have also introduced the *reflection functions* supported by $\mathcal{TOY}(\mathcal{FD})$. These constraints allow the user to recover information about the constraint solving process at runtime, and we have shown by examples how it is possible to construct user-defined labeling strategies via these constraints. These constraints provide a glass box approach to the $\mathcal{TOY}(\mathcal{FD})$ system that, to our knowledge, is new in the constraint functional logic programming.

Therefore, $\mathcal{TOY}(\mathcal{FD})$ combines the black box and glass box approaches. This is a contribution to pure FLP languages based on rewriting logic. As black box constraints (e.g., the *labeling* constraint) guarantee efficiency, the user should use them whenever possible and when the application allows them. Otherwise, the user is allowed to define, at a high level, specific constructs for particular applications.

This paper demonstrates that $\mathcal{TOY}(\mathcal{FD})$ allows a flexible modelling and quick prototyping at a very high level that cannot be reached by most of the existing constraint systems. $\mathcal{TOY}(\mathcal{FD})$ is freely available in [5].

References

1. H. Aït-kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993.
2. N. Barnier and P. Brisset. FaCiLe: a functional constraint library. *ALP Newsletter*, 14(2), May 2001.
3. A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In P. Wadler and V. Dahl, editors, *5th International Symposium on Practical Aspects of Declarative Languages (PADL'2003)*, number 2562 in LNCS, pages 320–338, New Orleans, Louisiana, USA, 2003. Springer.
4. A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD): Sketch of operational semantics. In F. Rossi, editor, *Principles and Practice of Constraint Programming (CP'2003)*, number 2833 in LNCS, pages 827–831, Kinsale, Ireland, 2003. Springer.
5. A. J. Fernández, T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD): System and user manual. Available at `http://toy.sourceforge.net/`, 2004.
6. Hanus M. (editor). PAKCS 1.4.0, User manual. The Portland Aachen Kiel Curry System. Available from `http://www.informatik.uni-kiel.de/~pakcs/`, 2002.
7. J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994.
8. T. Klove. Bounds and construction for difference triangle sets. *IEEE Transactions on Informatrion Theory*, 35:879–886, July 1989.
9. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, number 1631 in LNCS, pages 244–247, Trento, Italy, 1999. Springer.
10. J.-F. Puget. PECOS: A High Level Constraint Programming Language. In *Spicis'92*, Singapore, 1992.
11. M. Rodríguez-Artalejo. Functional and constraint logic programming. In H. Comon, C. Marché, and R. Trainen, editors, *Constraints in Computational Logics*, number 2002 in LNCS, pages 202–270. Springer, 2001. Revised Lectures of the International Summer School CCL'99.
12. J. Shearer. Some new optimum golomb rulers. *IEEE Transactions on Informatrion Theory*, 36:183–184, January 1990.
13. J. Shearer. Golomb ruler table. `www.research.ibm.com/people/s/shearer/-grtab.html`, 2004.
14. J. Siskind and D. McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In R. Fikes and W. Lehnert, editors, *Proceedings of the Eleventh National Conference on Artifcial Intelligence*, Menlo Park, California, 1993. AAAI Press.
15. P. Van Hentenryck. *The OPL optimization programming language*. The MIT Press, Cambridge, MA, 1999.
16. P. Van Roy, 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(6):717–763, November 2003.