# Using the $\mathcal{TOY}(\mathcal{FD})$ Constraint Functional Logic System

Antonio J. Fernández*

Dpto. de Lenguajes y Ciencias de la Computación
E.T.S.I.I., 29071 Teatinos
University of Málaga
Málaga, Spain
afdez@lcc.uma.es

Teresa Hortalá-González,
Fernando Sáenz-Pérez

Dpto. Sist. Informat. y Prog.
Univ. Complutense de Madrid
Av. Complutense, 28040 Madrid, Spain
{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.

The paper also introduces a novel proposal, in functional logic languages, to allow the recovering and management, at the user level, of internal information about the constraint solving process at runtime.
**keywords**: Functional Logic Programming, Finite Domain Constraints.

## 1 Introduction

In [1] we proposed the integration of finite domain (FD) constraints into the functional logic programming (FLP) language $\mathcal{TOY}$ [3] and, as a result, we presented the language $\mathcal{TOY}(\mathcal{FD})$ that integrates the best features of existing functional and logic languages, and FD constraint solving. Now, this paper illustrates, by means of examples, the features of $\mathcal{TOY}(\mathcal{FD})$, and shows its flexibility to solve combinatorial optimization problems.

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

*This author was partially supported by Spanish MCyT under contracts TIC2002-04498-C05-02 and TIN2004-7943-C04-01.

## 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 $\mathcal{TOY}$. Basically, $\mathcal{TOY}(\mathcal{FD})$ 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 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. A $\mathcal{TOY}(\mathcal{FD})$ program defines a set $FS$ of functions. Each $f \in FS$ has an associated principal type of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \tau$ (where $\tau$ does not contain $\rightarrow$). As usual in functional programming (FP), types are inferred and, optionally, can be declared in the program.

Basically, a $\mathcal{TOY}(\mathcal{FD})$ *program* $P$ is a set of *defining rules* for the function symbols in its signature. 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 pro-

gram 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 $p :: \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow$ `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 primitive function. Table 1 shows a small subset of the FD constraints supported by $\mathcal{TOY}(\mathcal{FD})$, where `int` is a predefined type for integers, and `[`$\tau$`]` is the type 'list of $\tau$'. The datatype `labelType` is a predefined type which is used to define the many search strategies for finite domain labeling [2].

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

Table 1: Some Predefined FD Constraints

$\mathcal{TOY}(\mathcal{FD})$ supports *relational constraints* including equality and disequality, *arithmetical constraints* including all the classical arithmetical 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), *a membership constraint* to restrict a list of variables to have values in an interval of integers, *propositional constraints* to define constraint reification, and *enumeration constraints* [1] with a number of options (including some for 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 [2] 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.

### 2.2 A Simple Programming Example

Below, a $\mathcal{TOY}(\mathcal{FD})$ program to solve the classical N-queen problem is shown.

```
include "misc.toy"
include "cflpfd.toy"

queens :: [labelType] -> int -> [int]
queens Label N = L <== length L==N, domain L 1 N,
                 constrain_all L,labeling Label L

constrain_all :: [int] ->  bool
constrain_all [] = true
constrain_all [X|Xs] = true <==
   constrain_between X Xs 1, constrain_all Xs

constrain_between :: int -> [int] -> int -> bool
constrain_between X [] N = true
constrain_between X [Y|Ys] N = true <==
   no_threat X Y N, N1 == N+1,
   constrain_between X Ys N1

no_threat:: int -> int -> int -> bool
no_threat X Y I = true <== X  #\= Y,
   X #+ I #\= Y, X #- I #\= Y
```

The intended meaning of the functions should be clear from their names and definitions, provided that `length L` returns the length of the list `L`, `domain L A B` constrains the domain of the elements in the list `L` to the closed integer interval [A,B], and `labeling S L` enumerates the variables in the list `L` following the search strategies specified in the list `S`.

[1] In general, constraint propagation is not enough to solve a constraint problem and, as a consequence, it is very frequent to employ an additional strategy called *labeling*, *enumeration* or *search* to solve it. Basically, *labeling* consists of selecting a variable, when no more constraint propagation is possible, to divide its domain and generate different computation branches in the search tree for further continuing with the propagation on each of the branches independently.

The first two lines are needed to include predefined functions in $\mathcal{TOY}(\mathcal{FD})$. Also, the first line in a function definition shows the type of its arguments followed by the type of its result, separated by `->`. An example of solving at the command prompt is shown below where `yes` stands for a `true` result.

```
TOY(FD)> queens [ff] 15 == L
  yes
  L ==  [1,3,5,14,11,4,10,7,13,15,2,8,6,9,12]
  Elapsed time: 0 ms.
```

## 3 Solving Constraint Problems

This section highlights the expressive power of $\mathcal{TOY}(\mathcal{FD})$ by proposing very expressive solutions to a number of constraint satisfaction problems. In general, all the examples described in this section show, among another features (e.g., constraint optimization, constraint composition, or curried notation) the combination of higher order (HO) applications and (indeterministic) lazy generation of constraints.

### 3.1 Optimization: the Golomb Ruler

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.

An *Optimal Golomb Ruler* (OGR) is defined as the shortest Golomb ruler for a number of marks, and 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 [4]. To date, the highest Golomb ruler whose shortest length is known is the ruler with 23 marks 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[2].

$\mathcal{TOY}(\mathcal{FD})$ 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 `X`). Below, we show a $\mathcal{TOY}(\mathcal{FD})$ program to solve OGRs with $N$ marks[3] and the solving of a goal for `N = 12`.

```
golomb :: int -> [int]
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

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]
      Elapsed time: 10918040 ms.
```

### 3.2 Laziness: Process Network

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

[2] http://www.distributed.net/ogr/. These solutions took several months to be found.

[3] We have proved that $\mathcal{TOY}(\mathcal{FD})$ 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, that is a reasonable time. For instance, the efficient constraint logic programming (CLP) system ECL$^i$PS$^e$ solves these instances in 287 and 75,300 seconds, respectively, although this of course depends on the problem model and the computation machine. See `http://www.icparc.ic.ac.uk/-eclipse/examples/golomb.ecl.txt`

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 defined 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 an example, and for the sake of simplicity, we can define them as follows:

```
process :: int -> int
process = (+3)

initial :: int
initial = 4

next :: int -> int
next = id   % Idempotence
```

However, this is not enough to produce an exit as a 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, server, answers` and `requests` as follows:

```
client :: int -> [int] -> [int]
client Ini Rs =
      [Ini | client (next (head Rs)) (tail Rs)]

server :: [int] -> [int]
server [P|Ps] = [process P| server Ps]

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` returns the head and tail of a list respectively. Below, we show an example of solving that evaluates exactly the first 15th requests.

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

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 make use of the constraint facilities of the language.

### 3.3 Pipelines

Pipelines can be a powerful tool to solve heterogeneous constraint satisfaction problems, and they are easily expressed at a high level in $\mathcal{TOY}(\mathcal{FD})$ via applying HO constraints and curried notation. For example, consider the OGR, N-queens and Client-Server $\mathcal{TOY}(\mathcal{FD})$ 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 3.2 as *process = (+1)*. The solving of this goal, as in the preceding example of client-server architecture, generates N answers in the form of an N-elements list `A` from an initial request 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 solve the OGR problem with $\mathtt{a}_i$ marks, returning a new list `S` containing `N` solutions for OGRs with marks $\mathtt{a}_1,\ldots,\mathtt{a}_N$. Finally, for each solution $\mathtt{s}_{\mathtt{a}_i}$ `=` $[\mathtt{o}_1,\ldots,\mathtt{o}_{\mathtt{a}_i}]$ to the OGR with $\mathtt{a}_i$ marks in `S`, the first solution of the $\mathtt{o}_k$-queens problem (with `k` $\in$ `{1,...,`$\mathtt{a}_i$`}`) is calculated.

For example, the goal shown above (for `N = 2`) first calculates the solutions for the OGR with 4 and 5 marks (i.e., `[0,1,4,6]` and `[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.

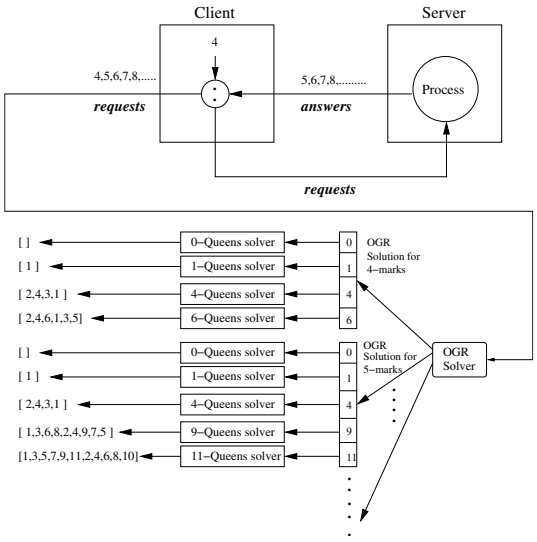


Figure 1: Pipeline with Client-Server Architecture

```
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 ]]]
  Elapsed time: 31 ms.
```

This example illustrates how easy and natural may be the combination of different problems in $\mathcal{TOY}(\mathcal{FD})$ [4]. This level of expressiveness cannot be reached in a CLP(FD) system.

## 4 Reflection Functions

$\mathcal{TOY}(\mathcal{FD})$ is a system that combines efficient *black box constraints* (i.e., those already shown that use specialized propagation mechanisms for the FD constraints, leading thus to a major efficiency) and *glass box constraints* (i.e., those that allow the user to define new constraints in terms of primitive constraints). To our knowledge, this is the first time that a pure constraint FLP language provides this capability.

### 4.1 Recovering Internal Information at Runtime

In $\mathcal{TOY}(\mathcal{FD})$, the glass box approach is based on a set of predefined functions called *reflection constraints* 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 constraints or even new search strategies. Below, we show part of this set of reflection constraints [2]:

```
fd_var :: int -> bool
fd_min :: int -> int
fd_max :: int -> int
fd_size :: int -> int
fd_degree :: int -> int
fd_neighbors :: int -> [int]

empty_interval :: int -> int -> bool
fd_set :: int -> [fdset] -> bool
is_fdset :: [fdset] -> bool
empty_fdset :: [fdset] -> bool
fdset_size ::  [fdset] -> int
fdset_min ::  [fdset] -> int
fdset_add_element ::  [fdset] -> int -> [fdset]
fdset_del_element ::  [fdset] -> int -> [fdset]
fdset_intersection :: [fdset] -> [fdset] -> [fdset]
fdset_member ::  int -> [fdset] -> bool
fdset_equal :: [fdset] -> [fdset]-> bool
fdset_subset :: [fdset] -> [fdset]-> bool
fdset_subtract :: [fdset] -> [fdset] -> [fdset]

fdset_union :: [fdset] -> [fdset] -> [fdset]
fdset_complement :: [fdset] -> [fdset]
fdset_belongs :: int -> int -> bool
```

`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` is the cardinality of the domain of variable `X`, and `fd_neighbors X` calculates 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 a FD variable and its domain is not a singleton value.

[4] Note that no additional $\mathcal{TOY}(\mathcal{FD})$ code is required, except that formulating the solutions to the N-queens and golomb ruler problems, and the goal shown above completely captures the architecture depicted in Figure 1.

Other constraints return specific information about the domains. For instance, `empty_interval Min Max` is true if the interval `[Min,Max]` is not empty and `fd_set X Dom` is true if `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`. The meaning of the rest should be clear from their names.

### 4.2 Programmable Search

As an example of using reflection functions, here we program one of the most popular labeling strategies, the so-called *first fail* that basically selects the variable with the least number of values in its domain, and it is often supported by existing constraint systems[5]. See below a user defined $\mathcal{TOY}(\mathcal{FD})$ function `labelff/1` that implements it.

```
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 the function `choose_min/3`) by making use of information recovered by reflection constraints `fd_set/2` and `fdset_size/1`). Also, when there is just one variable `X`, it reactivates the search process by dividing `X`'s domain by the value `(fd_min X)`.

[5] $\mathcal{TOY}(\mathcal{FD})$ also provides it by making use of the LabelType value `ff` in the `labeling` constraint for this strategy as shown in the example of Section 2.2.

## 5 Conclusions

This paper demonstrates, by examples, the potential of $\mathcal{TOY}(\mathcal{FD})$, a functional logic language that integrates FD constraint solving, lazy evaluation, higher order applications of functions and constraints, polymorphism, type checking, 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.

We have also introduced the *reflection constraints* supported by $\mathcal{TOY}(\mathcal{FD})$, that are new in the constraint functional logic programming arena, and have shown by examples how it is possible to construct new user-defined constraint constructs (e.g., labeling strategies) via these constraints. $\mathcal{TOY}(\mathcal{FD})$ is freely available in [2].

## References

[1] A. J. Fernández, M. T. Hortalá-González, and F. Sáenz-Pérez. Solving combinatorial problems with a constraint functional logic language. In *PADL'2003*, number 2562 in LNCS, pages 320–338, New Orleans, 2003. Springer.

[2] A. J. Fernández, T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD): System and user manual. Available at `http://toy.sourceforge.net/`, 2004.

[3] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *RTA'99*, number 1631 in LNCS, pages 244–247, Trento, Italy, 1999. Springer-Verlag.

[4] J. Shearer. Some new optimum golomb rulers. *IEEE Transactions on Informatrion Theory*, 36:183–184, January 1990.