Solving Combinatorial Problems with a Constraint Functional Logic Language

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

[1] Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain***
[2] Depto. de Sistemas Informáticos y Programación Universidad Complutense de Madrid, Spain†
afdez@lcc.uma.es, {teresa,fernan}@sip.ucm.es

Abstract. This paper describes a proposal to incorporate finite domain constraints in a functional logic system. The proposal integrates functions, higher-order patterns, partial applications, non-determinism, logical variables, currying, types, lazyness, domain variables, constraints and finite domain propagators.
The paper also presents TOY(FD), an extension of the functional logic language TOY that provides FD constraints, and shows, by examples, that TOY(FD) combines the power of constraint logic programming with the higher-order characteristics of functional logic programming.

Keywords: Constraints, Functional Logic Programming, Finite Domains.

1 Introduction

Constraint logic programming (CLP) emerged recently to increase both the expressiveness and efficiency of logic programming (LP) [8]. The basic idea in CLP consists of replacing the classical LP unification by constraint solving on a given computation domain. Among the domains for CLP, the finite domain (FD) [11] is one of the most and best studied since it is a suitable framework for solving discrete constraint satisfaction problems.

Unfortunately, literature lacks proposals to integrate FD constraints in functional programming (FP). This seems to be caused by the relational nature of FD constraints that do not fit well in FP. To overcome this limitation we consider a functional logic programming (FLP) setting [7] and integrate FD constraints in the FLP language TOY [9] giving rise to CFLP(FD) (i.e., constraint functional logic programming over finite domains).

This paper describes, to our knowledge, the first FLP system that completely incorporates FD constraints. The main contribution then is to show how

*** 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.
† Hortalá-González and Fernando Sáenz-Pérez were supported by the Spanish project PR 48/01-9901 funded by UCM.

V. Dahl and P. Wadler (Eds.): PADL 2003, LNCS 2562, pp. 320–338, 2003.

to apply FD constraints to a functional logic language. We believe that our proposal has many advantages and considerable potential since it benefits from both the logical and functional settings and the constraint framework, by first taking functions, higher-order patterns, partial applications, non-determinism, logical variables, currying, function composition, types and lazy evaluation from functional logic programming, and, second, domain variables, constraints and efficient propagators from finite domain constraint programming.

The paper is structured as follows. Section 2 presents a formalization for CFLP(FD). Section 3 describes briefly TOY(FD), an implementation of a CFLP(FD) system, whereas Section 4 shows several examples of programming in TOY(FD). Then, Section 5 discusses some related work and, Section 6 develops a performance comparison with related systems. Finally, the paper ends with some indications for further research and some conclusions.

2 CFLP(FD) Programs

This section presents, by following the formalization given in [6], the basics about syntax, type discipline, and declarative semantics of CFLP(FD) programs.

2.1 CFLP(FD) Fundamental Concepts

Types and Signatures: We assume a countable set $\mathcal{TV}ar$ of *type variables* $\alpha,\ \beta,\ \ldots$ and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* $C \in TC^n$. Types $\tau \in \mathit{Type}$ have the syntax

$$\tau ::= \alpha \quad \mid C\ \tau_1 \ldots \tau_n \quad \mid \tau \to \tau' \mid (\tau_1, \ldots, \tau_n)$$

By convention, $C\ \overline{\tau}_n$ abbreviates $C\ \tau_1 \ldots \tau_n$, "$\to$" associates to the right, $\overline{\tau}_n \to \tau$ abbreviates $\tau_1 \to \cdots \to \tau_n \to \tau$, and the set of type variables occurring in τ is written $tvar(\tau)$. A type without any occurrence of "$\to$" is called a *datatype*. The type $(\tau_1, \ldots, \tau_n)$ is intended to denote n-tuples. FD variables are integer variables. A *signature* over TC is a triple $\Sigma = \langle TC,\ DC,\ FS\rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*. Each n-ary $c \in DC^n$ comes with a principal type declaration $c\ ::\ \overline{\tau}_n \to C\ \overline{\alpha}_k$, where $n, k \geq 0, \alpha_1, \ldots, \alpha_k$ are pairwise different, τ_i are datatypes, and $tvar(\tau_i) \subseteq \{\alpha_1,\ \ldots,\ \alpha_k\}$ for all $1 \leq i \leq n$. Also, every n-ary $f \in FS^n$ comes with a principal type declaration $f\ ::\ \overline{\tau}_n \to \tau$, where τ_i, τ are arbitrary types. In practice, each CFLP(FD) program P has a signature which corresponds to the type declarations occurring in P. For any signature Σ, we write $\Sigma_\perp$ for the result of extending Σ with a new data constructor $\perp\ ::\ \alpha$, intended to represent an undefined value that belongs to every type. As notational conventions, we use $c \in DC$, $f, g \in FS$ and $h \in DC \cup FS$.

FD constraints: A *FD constraint* is a primitive function declared with type either

Table 1. Datatypes for FD Constraints

```
data labelType = ff | ffc | leftmost | mini | maxi | step | enum | bisect | up
               | down | all | toMinimize int | toMaximize int | assumptions int
data statistics = resumptions | entailments | prunings | backtracks | constraints
data reasoning = value | domains | range
data options = on reasoning | complete bool
data typeprecedence = d (int,int,int)
data newOptions = precedences [typeprecedence] | path_consistency bool
                | static_sets bool | edge_finder bool | decomposition bool
```

Table 2. Some FD Constraints in TOY(FD)

RELATIONAL	ARITHMETICAL
(#>) :: int → int → bool	(#∗) :: int → int → int
(#<) :: int → int → bool	(#/) :: int → int → int
(#>=) :: int → int → bool	(#+) :: int → int → int
(#<=) :: int → int → bool	(#−) :: int → int → int
(# =) :: int → int → bool	sum :: [int] → (int → int → bool) → int → bool
(#\=) :: int → int → bool	scalar_product :: [int] → [int] → (int → int → bool) → int → bool
COMBINATORIAL	
assignment :: [int] → [int] → bool	all_different :: [int] → bool
circuit :: [int] → bool	all_different' :: [int] → [options] → bool
circuit' :: [int] → [int] → bool	serialized :: [int] → [int] → bool
all_distinct :: [int] → bool	serialized' :: [int] → [int] → [newOptions] → bool
all_distinct' :: [int]→[options]→bool	cumulative :: [int] → [int] → [int] → int → bool
exactly :: int → [int] → int → bool	cumulative' :: [int] → [int] → [int] → int → [newOptions] → bool
element :: int → [int] → int → bool	count :: int → [int] → (int → int → bool) → int → bool
MEMBERSHIP	
domain :: [int] → int → int → bool	
ENUMERATION	**STATISTICS**
labeling :: [labelType]→[int]→bool	fd_statistics :: statistics → int → bool
indomain :: int → bool	fd_statistics' :: bool

- `int` → `int` → `int` to transform pairs of FD variables into FD variables, or
- $\overline{\tau}_n$ → `bool` such that for all τ_i in $\overline{\tau}_n$, $\tau_i \in \mathit{Type}_{FD}$ and $\mathit{Type}_{FD} \subset \mathit{Type}$ is

$$\mathit{Type}_{FD} = \{\ \texttt{int}, [\texttt{int}], [\texttt{labelType}], \texttt{statistics}, (\texttt{int} \to \texttt{int} \to \texttt{bool}), [\texttt{options}], [\texttt{newOptions}]\ \}.$$

`int` is a predefined type for integers, and $[\tau]$ is the type 'list of τ'. The datatypes `labelType, statistics, options` and `newOptions` are predefined types and their complete definitions are shown in Table 1.

Some FD constraints supported in our language are shown in Table 2. Examples of the first sort of constraints are the arithmetic functions #+, #−, #∗ and #/. Examples of the second sort of constraints are the relations #<, and #> as well as the functions `all_distinct'`/2, and `labeling`/2.

In the rest of the section, $FS_{FD} \subset FS^n$ denotes the set of FD constraints that return a Boolean value.

In CFLP(FD), functions (e.g., constraints) are first-class citizens, which means that a function can appear in any place where a data can. As a direct consequence, a 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*. As examples of HO constraints, look at the FD constraints `sum`/3, `scalar_product`/4 and `count`/4 in Table 2. These constraints accept a FD constraint of type `int` → `int` → `bool` (e.g., #<, or #>) as argument.

Expressions and Patterns: In the sequel, we always assume a given signature Σ, often not made explicit in the notation. Assuming a countable set $\mathcal{V}ar$ of (data) variables $X, Y, \ldots$ disjoint from $\mathcal{TV}ar$ and Σ, *partial expressions* $e \in Exp_\bot$ have the syntax

$$e ::= \bot \mid X \mid h \mid e\ e' \mid (e_1, \ldots, e_n)$$

where $X \in \mathcal{V}ar$, $h \in DC \cup FS$. Expressions of the form $e\,e'$ stand for the application of expression e (playing as a function) to expression e' (playing as an argument), while expressions $(e_1, \ldots, e_n)$ represent tuples with n components. As usual, we assume that application associates to the left and thus $e_0\ e_1 \ldots\ e_n$ abbreviates $(\ldots(e_0\ \ e_1)\ldots)\ \ e_n$. The set of data variables occurring in e is written $var(e)$.

An expression e is called *linear* iff every $X \in var(e)$ has one single occurrence in e. 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 $c \in DC^n$ $(n \geq 0)$ and some n-tuple of expressions $\overline{e}_n = (e_1, \ldots, e_n)$ where e_i is in head normal form.

Partial patterns $t \in Pat_\bot \subset Exp_\bot$ are built as

$$t ::= \bot \mid X \mid c\ t_1\ \ldots\ t_m \mid f\ t_1\ \ldots\ t_m$$

where $X \in \mathcal{V}ar$, $c \in DC^k$, $0 \leq m \leq k$, $f \in FS^n$, $0 \leq m < n$ and $t_i \in Pat_\bot$ for all $1 \leq i \leq m$. They represent *approximations* of the values of expressions. Partial patterns of the form $f\ t_1 \ldots t_m$ with $f \in FS^n$ and $m < n$ serve as a convenient representation of functions as values [6]; therefore functions becoming first-class citizens of the language. Expressions and patterns without any occurrence of $\bot$ are called *total*. The sets of total expressions and patterns are denoted, respectively, by Exp and Pat. Actually, the symbol $\bot$ never occurs in a program's text.

Substitutions: A *substitution* is a mapping $\theta : \mathcal{V}ar \to Pat$ with a unique extension $\hat{\theta} : Exp \to Exp$, which is also denoted as θ. As usual, $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ stands for the substitution with domain $\{X_1, \ldots, X_n\}$ which

satisfies $\theta(X_i) = t_i$ for all $1 \leq i \leq n$. *Subst* denotes the set of all variable substitutions.

Up to this point we have considered *data substitutions*. *Type substitutions* can be defined similarly, as mappings $\theta_t : \mathcal{TV}ar \rightarrow Type$ with a unique extension $\hat{\theta}_t : Type \rightarrow Type$, also denoted θ_t. *TSubst* denotes the set of all type substitutions.

Finite Domains: A *finite domain* (FD) is a mapping $\delta : \mathcal{V}ar \rightarrow \wp(Integer)$ (as usual $\wp(C)$ denotes the powerset of the set C), with a unique extension $\hat{\delta} : Exp \rightarrow Exp$, which will be denoted also as δ, and *Integer* is the set of integers. We use $\delta = \{X_1 \in d_1, \ldots, X_n \in d_n\}$, which stands for the FD with domain $\{X_1, \ldots, X_n\}$ and satisfies $\delta(X_i) = d_i$ for all $1 \leq i \leq n$, where $d_i \subseteq Integer$.

By convention, if δ is either a FD or a substitution we write $e\delta$ instead of $\delta(e)$, and $\delta\sigma$ for the composition of δ and σ s.t. $e(\delta\sigma) = (e\delta)\sigma$ for any e.

2.2 Well-Typedness

Inspired by Milner's type system we now introduce the notion of well-typed expression. We define a *type environment* as any set T of type assumptions $X :: \tau$ for data variables s.t. T does not include two different assumptions for the same variable. The *domain* $dom(T)$ of a type environment is the set of all data variables that occur in T. For any variable $X \in dom(T)$, the unique type τ s.t. $(X :: \tau) \in T$ is denoted as $T(X)$. The notation $(h :: \tau) \in_{var} \Sigma$ is used to indicate that Σ includes the type declaration $h :: \tau$ up to a renaming of type variables. *Type judgements* $(\Sigma, T) \vdash_{WT} e :: \tau$ are derived by means of the following *type inference* rules:

VR $(\Sigma, T) \vdash_{WT} X :: \tau$, if $T(X) = \tau$.
ID $(\Sigma, T) \vdash_{WT} h :: \tau\sigma_t$, if $(h :: \tau) \in_{var} \Sigma_\perp,\ \sigma_t \in TSubst$.
AP $(\Sigma, T) \vdash_{WT} (e\ e_1) :: \tau$, if $(\Sigma, T) \vdash_{WT} e :: (\tau_1 \rightarrow \tau)$, $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1$, for some $\tau_1 \in Type$.
TP $(\Sigma, T) \vdash_{WT} (e_1, \ldots, e_n) :: (\tau_1, \ldots, \tau_n)$, if $\forall i \in \{1, \ldots, n\} : (\Sigma, T) \vdash_{WT} e_i :: \tau_i$.

An expression $e \in Exp_\perp$ is called *well-typed* iff there exist some *type environment* T and some type τ, s.t. the *type judgement* $T \vdash_{WT} e :: \tau$ can be derived. Expressions that admit more than one type are called *polymorphic*. A well-typed expression always admits a so-called *principal type* (PT) that is more general than any other. A pattern whose PT determines the PTs of its subpatterns is called *transparent*.

A *well-typed CFLP(FD) program* P is a set of *well-typed defining rules* for the function symbols in its signature. Defining rules for $f \in FS^n$ with principal type declaration $f :: \overline{\tau}_n \rightarrow \tau$ have the form

$$\text{(R)} \quad \underbrace{f\ t_1\ \ldots\ t_n}_{\text{left hand side}} \ = \ \underbrace{r}_{\text{right hand side}} \ \Leftarrow \ \underbrace{C}_{\text{Condition}}$$

and must satisfy the following requirements:

1. $t_1 \ldots t_n$ is a linear sequence of transparent patterns and r is an expression.
2. The *condition* C is a sequence of *conditions* $C_1, \ldots, C_k$, where each C_i can be either a *joinability statement* of the form $e == e'$, or a *disequality statement* of the form $e /= e'$, with $e, e' \in \mathit{Exp}$, or a Boolean function g of the form $g\ e_1 \ldots\ e_m$, with $e_i \in \mathit{Exp}$ and $g \in FS^m$ (of course, perhaps $g \in FS_{FD}$).
3. There exists some type environment T with domain $var(R)$ which well-types the defining rule in the following sense:
   a) For all $1 \leq i \leq n$: $(\Sigma, T) \vdash_{WT} t_i :: \tau_i$.
   b) $(\Sigma, T) \vdash_{WT} r :: \tau$.
   c) For each $(e == e') \in C$, $\exists \mu \in \mathit{Type}$ s.t. $(\Sigma, T) \vdash_{WT} e :: \mu :: e'$.
   d) For each $(e/= e') \in C$, $\exists \mu \in \mathit{Type}$ s.t. $(\Sigma, T) \vdash_{WT} e :: \mu :: e'$.
   e) For each $(g\ e_1 \ldots\ e_m) \in C$, where $g :: \tau_1 \to \ldots \to \tau_m \to \texttt{bool}$, $(\Sigma, T) \vdash_{WT} e_i :: \tau_i$, and $\tau_i \in \mathit{Type}$, for all $1 \leq i \leq m$.

Here, $(\Sigma, T) \vdash_{WT} a :: \tau,\ (\Sigma, T) \vdash_{WT} b :: \tau$ denotes $(\Sigma, T) \vdash_{WT} a :: \tau :: b$.

Informally, the intended meaning of a program rule as (R) above is that a call to a function f can be reduced to r whenever the actual parameters match the patterns t_i, and both the joinability conditions, the disequality conditions and the Boolean functions (including the FD constraints) are satisfied. A condition $e == e'$ is satisfied by evaluating e and e' to some common total pattern. Predicates are viewed as a particular kind of functions, with type $p :: \overline{\tau}_n \to \texttt{bool}$. 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 $p\ t_1\ \ldots\ t_n = \mathit{true} \Leftarrow C_1, \ldots, C_k$.

A *well-typed goal* G has the same form as a well-typed expression and must satisfy the admissibility requirements but regarding the empty set of variables.

In general, a CFLP(FD) system is expected to *solve* goals, returning a set of 4-tuples $\langle E, \sigma, C, \delta \rangle$ as a computed answer where $E \in \mathit{Exp}$ is a TOY expression, $\sigma \subseteq \mathit{Subst}$ is the set of variable substitutions, C is a set of disequality constraints, and δ is the set of pruned finite domains.

3 TOY(FD): A CFLP(FD) Implementation

This section describes briefly part of TOY(FD), our CFLP(FD) implementation that extends the TOY system [9] to deal with FD constraints.

Table 2 shows some FD constraints provided by TOY(FD). Among others, TOY(FD) supports equality and disequality constraints, well-known global constraints (e.g., `all_different`/1), a membership constraint (i.e., `domain`/3) and enumeration constraints (e.g., `labeling`/2) with a number of options to reactivate the search process when no more constraint propagation is possible.

TOY(FD) also provides a set of constraints (not shown in Table 2), called *reflection constraints*, that allow to recover information about constrained FD variables and their associated domains during the solving of a goal (e.g., the reflection constraints `fd_min`, `fd_max` :: `int` $\to$ `int` applied to a FD variable return respectively the minimum and maximum value of this FD variable in its current

domain). For reasons of space, we do not describe all the constraints in detail and encourage the interested reader to visit the link proposed in [4] for a more detailed explanation (this link also shows several examples of TOY(FD) programs).

TOY(FD) is implemented on top of Sicstus Prolog 3.8.4 and uses the FD constraint solver of SICStus [2]. FD constraints are integrated in TOY(FD) as functions and evaluated internally by using mainly two predicates: `hnf(E,H)`, which specifies that `H` is one of the possible results of narrowing the expression `E` into head normal form, and `solve`/1, which checks the satisfiability of constraints (of rules and goals) previously to the evaluation of a given rule. This predicate is, basically, defined as follows[1]:

(1) $\texttt{solve}((\varphi, \varphi')) \quad :- \texttt{solve}(\varphi), \texttt{solve}(\varphi').$
(2) $\texttt{solve}(\texttt{L} == \texttt{R}) \quad :- \texttt{hnf}(\texttt{L}, \texttt{L}'), \texttt{hnf}(\texttt{R}, \texttt{R}'), \texttt{equal}(\texttt{L}', \texttt{R}').$
(3) $\texttt{solve}(\texttt{L} \,/= \texttt{R}) \quad :- \texttt{hnf}(\texttt{L}, \texttt{L}'), \texttt{hnf}(\texttt{R}, \texttt{R}'), \texttt{notequal}(\texttt{L}', \texttt{R}').$
(4) $\texttt{solve}(\texttt{L}\#\diamondsuit\ \texttt{R}) \quad :- \texttt{hnf}(\texttt{L}, \texttt{L}'), \texttt{hnf}(\texttt{R}, \texttt{R}'), \{\texttt{L}'\#\diamondsuit\texttt{R}'\}.$
where $\diamondsuit \in \{<, <=, >, >=, =, \backslash=\}$.
(5) $\texttt{solve}(\texttt{C}\ \texttt{A}_1 \ldots \texttt{A}_\texttt{n}) :- \texttt{hnf}(\texttt{A}_1, \texttt{A}'_1), \ldots, \texttt{hnf}(\texttt{A}_\texttt{n}, \texttt{A}'_\texttt{n}), \{\texttt{C}(\texttt{A}'_1, \ldots, \texttt{A}'_\texttt{n})\}.$
where `C` is any constraint returning a Boolean.

The interaction with SICStus FD constraint solver is reflected in the two last clauses: every time a FD constraint appears, the solver is eventually invoked with a goal $\{G\}$ where G is the translation of the FD constraint from TOY(FD) to SICStus Prolog. The expressions have to be 'simplified' in order to allow the solver to solve the constraint. By simplifying we mean computing the head normal forms (hnf) of both expressions.

4 Programming in TOY(FD)

Any CLP(FD)-program can be straightforwardly translated into a CFLP(FD)-program. As example, Section 4.1 shows the TOY (FD) code to solve the classical arithmetic puzzle "send+more=money". We do not insist more on this matter, but prefer to concentrate on the extra capabilities of the language and illustrate some of them by means of a more interesting example developed in Section 4.2.

4.1 An Introductory TOY(FD) Example

Below, a TOY(FD) program to solve the classical arithmetic puzzle "send more money" is shown. TOY(FD) allows to use infix constraint operators such as `#>` to build the expression `X #> Y`, which is understood as `#> X Y`. The signature of the program can be easily inferred from the type declarations included in its text. The intended meaning of the functions should be clear from their names, definitions and Tables 1 and 2.

[1] The code does not correspond exactly to the implementation, which is the result of many transformations and optimizations.

```
smm :: int -> int -> int -> int -> int -> int -> int
           -> [labelType] -> bool
smm S E N D M O R Y Label :- domain [S,E,N,D,M,O,R,Y] 0 9,
            S #> 0, M #> 0,
            all_different [S,E,N,D,M,O,R,Y],
                            1000#*S #+ 100#*E #+ 10#*N #+ D
                         #+ 1000#*M #+ 100#*O #+ 10#*R #+ E
            #=  10000#*M #+ 1000#*O #+ 100#*N #+ 10#*E #+ Y,
            labeling Label [S,E,N,D,M,O,R,Y]
```

4.2 A Hardware Design Problem

A more interesting example comes from the hardware area. In this setting, many constrained optimization problems arise in the design of both sequential and combinational circuits as well as the interconnection routing between components. Constraint programming has been shown to effectively attack these problems. In particular, the interconnection routing problem (one of the major tasks in the physical design of very large scale integration - VLSI - circuits) have been solved with constraint logic programming [13].

For the sake of conciseness and clarity, we focus on a constraint combinational hardware problem at the logical level but adding constraints about the physical factors the circuit has to meet. This problem will show some of the nice features of TOY for specifying issues such as behavior, topology and physical factors.

Our problem can be stated as follows. Given a set of gates and modules, a switching function, and the problem parameters maximum circuit area, power dissipation, cost, and delay (dynamic behavior), the problem consists of finding possible topologies based on the given gates and modules so that a switching function and constraint physical factors are met. In order to have a manageable example, we restrict ourselves to the logical gates NOT, AND, and OR. We also consider circuits with three inputs and one output, and the physical factors aforementioned. We suppose also the following problem parameters:

Gate	Area	Power	Cost	Delay
NOT	1	1	1	1
AND	2	2	1	1
OR	2	2	2	2

In the sequel we will introduce the problem by first considering the features TOY offers for specifying logical circuits, what are its weaknesses, and how they can effectively be solved with the integration of constraints in TOY(FD) .

Example 1. FLP Simple Circuits. With this example we show the FLP approach that can be followed for specifying the problem stated above. We use patterns to provide an *intensional* representation of functions. The alias `behavior` is used for representing the type `bool` → `bool` → `bool` → `bool`. Functions of this type are intended to represent simple circuits which receive three Boolean inputs and return a Boolean output. Given the Boolean functions `not`, `and`, and `or` defined elsewhere, we specify three-input, one-output simple circuits as follows.

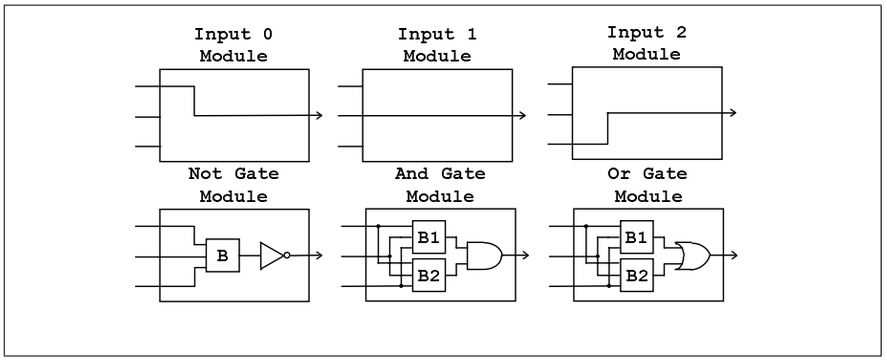

Fig. 1. Basic Modules.

```
i0,i1,i2 :: behavior
i0 I2 I1 I0 = I0
i1 I2 I1 I0 = I1
i2 I2 I1 I0 = I2

notGate :: behavior -> behavior
notGate B I2 I1 I0 = not (B I2 I1 I0)

andGate, orGate :: behavior -> behavior -> behavior
andGate B1 B2 I2 I1 I0 = and (B1 I2 I1 I0) (B2 I2 I1 I0)
orGate B1 B2 I2 I1 I0 = or (B1 I2 I1 I0) (B2 I2 I1 I0)
```

Functions `i0`, `i1`, and `i2` represent inputs to the circuits, that is, the minimal circuit which just copies one of the inputs to the output. (In fact, this can be thought as a fixed multiplexer - selector.) They are combinatorial modules as depicted in Figure 1. The function `notGate` outputs a Boolean value which is the result of applying the NOT gate to the output of a circuit of three inputs. In turn, functions `andGate` and `orGate` output a Boolean value which is the result of applying the AND and OR gates, respectively, to the outputs of three inputs-circuits (see Figure 1).

These functions can be used in a higher-order fashion just to generate or match topologies. In particular, the higher-order functions `notGate`, `andGate` and `orGate` take behaviors as parameters and build new behaviors, corresponding to the logical gates NOT, AND and OR. For instance, the multiplexer depicted in Figure 2 can be represented by the following pattern:

```
orGate (andGate i0 (notGate i2)) (andGate i1 i2).
```

This first-class citizen higher-order pattern can be used for many purposes. For instance, it can be compared to another pattern or it can be applied to actual values for its inputs in order to compute the circuit output. So, with the previous pattern, the goal:

```
P == orGate (andGate i0 (notGate i2)) (andGate i1 i2),
P true false true == O
```

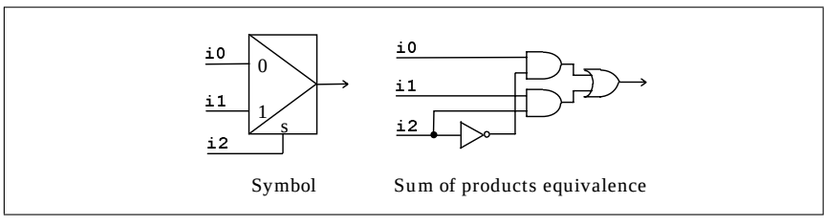

Fig. 2. Two-Input Multiplexer Circuit.

is evaluated to `true` and produces the substitution `O == false`. The rules that define the behavior can be used to generate circuits, which can be restricted to satisfy some conditions. If we use the standard arithmetics, we could define the following set of rules for computing or limiting the power dissipation.

```
power :: behavior -> int
power i0 = 0
power i1 = 0
power i2 = 0
power (notGate C) = notGatePower + (power C)
power (andGate C1 C2) = andGatePower + (power C1) + (power C2)
power (orGate C1 C2) = orGatePower + (power C1) + (power C2)
```

Then, we can submit the goal `power B == P, P < maxPower` (provided the function `maxPower` acts as a problem parameter that returns just the maximum power allowed for the circuit) in which the function `power` is used as a behavior generator[2]. As outcome, we get several solutions ($\langle$`i0`, {`P==0`}, {}, {}$\rangle$, $\langle$`i1`, {`P==0`}, {}, {}$\rangle$, $\langle$`i2`, {`P==0`}, {}, {}$\rangle$, $\langle$`not i0`, {`P==1`}, {}, {}$\rangle$, ... , $\langle$`not (not i0)`, {`P==2`}, {}, {}$\rangle$, ... Declaratively, it is fine; but our operational semantics requires a head normal form for the application of the arithmetic operand `+`. This implies that we reach no more solutions beyond $\langle$ `not (` ... `(not i0)` ... `)`, `maxPower`, {}, {}$\rangle$ because the application of the fourth rule of `power` yields to an infinite computation. This drawback is solved by recursing to Peano's arithmetics, that is:

```
data nat = z | s nat

power' :: behavior -> nat
power' i0 = z
power' i1 = z
power' i2 = z
power' (notGate C) =
    plus notGatePower (power' C)

plus :: nat -> nat -> nat
plus z Y = Y
plus (s X) Y = s (plus X Y)

less :: nat -> nat -> bool
less z (s X) = true
less (s X) (s Y) = less X Y
```

[2] Equivalently and more concisely, `power B` $<$ `maxPower` could be submitted, but doing so we make the power unobservable.

```
power' (andGate C1 C2) =
    plus andGatePower (plus (power' C1) (power' C2))
power' (orGate C1 C2) =
    plus orGatePower  (plus (power' C1) (power' C2))
```

So, we can submit the goal `less (power' P) (s (s (s z)))`, where we have written down explicitly the maximum power (3 power units).

With the second approach we get a more awkward representation due to the use of successor arithmetics. The first approach to express this problem is indeed more declarative than the second one, but we get non-termination. FD constraints can be profitably applied to the representation of this problem as we show in the next example.

Example 2. CFLP(FD) Simple Circuits
As for any constraint problem, modelling can be started by identifying the FD constraint variables. Recalling the problem specification, circuit limitations refer to area, power dissipation, cost, and delay. Provided we can choose finite units to represent these factors, we choose them as problem variables. A circuit can therefore be represented by the 4-tuple state ⟨area, power, cost, delay⟩. The idea to formulate the problem consists of attaching this state to an ongoing circuit so that state variables reflect the current state of the circuit *during* its generation. By contrast with the first example, we do not "generate" and then "test", but we "test" when "generating", so that we can find failure in advance. A domain variable has a domain attached indicating the set of possible assignments to the variable. This domain can be reduced during the computation. Since domain variables are constrained by limiting factors, during the generation of the circuit a domain may become empty. This event prunes the search space avoiding to explore a branch known to yield no solution. Let's firstly focus on the area factor. The following function generates a circuit characterized by its state variables.

```
type area, power, cost, delay = int
type state = (area, power, cost, delay)
type circuit = (behavior, state)
genCir :: state -> circuit
genCir (A, P, C, D) = (i0, (A, P, C, D))
genCir (A, P, C, D) = (i1, (A, P, C, D))
genCir (A, P, C, D) = (i2, (A, P, C, D))
genCir (A, P, C, D) = (notGate B, (A, P, C, D)) <==
             domain [A] ((fd_min A) + notGateArea)  (fd_max A),
             genCir (A, P, C, D) == (B, (A, P, C, D))
genCir (A, P, C, D) = (andGate B1 B2, (A, P, C, D)) <==
             domain [A] ((fd_min A) + andGateArea)  (fd_max A),
             genCir (A, P, C, D) == (B1, (A, P, C, D)),
             genCir (A, P, C, D) == (B2, (A, P, C, D))
genCir (A, P, C, D) = (orGate B1 B2, (A, P, C, D)) <==
             domain [A] ((fd_min A) + orGateArea)  (fd_max A),
```

```
genCir (A, P, C, D) == (B1, (A, P, C, D)),
genCir (A, P, C, D) == (B2, (A, P, C, D))
```

The function `genCir` has an argument to hold the circuit state and returns a circuit characterized by a behavior and a state. (Note that we can avoid the use of the state tuple as a parameter, since it is included in the result.) The template of this function is like the previous example. The difference lies in that we perform domain pruning during circuit generation with the membership constraint `domain`, so that each time a rule is selected, the domain variable representing area is reduced in the size of the gate selected by the operational mechanism. For instance, the circuit area domain is reduced in a number of `notGateArea` when the rule for `notGate` has been selected. For domain reduction we use the reflection functions `fd_min` and `fd_max`. This approach allows us to submit the following goal:

```
domain [Area] 0 maxArea,
genCir (Area, Power, Cost, Delay) == Circuit
```

which initially sets the possible range of area between 0 and the problem parameter area expressed by the function `maxArea`, and then generates a `Circuit`. Recall that testing is performed during search space exploration, so that termination is ensured because the add operation is monotonic. The mechanism which allows this "test" when "generating" is the set of propagators, which are concurrent processes that are triggered whenever a domain variable is changed (pruned). The state variable delay is more involved since one cannot simply add the delay of each function at each generation step. The delay of a circuit is related to the maximum number of levels an input signal has to traverse until it reaches the output. This is to say that we cannot use a single domain variable for describing the delay. Therefore, considering a module with several inputs, we must compute the delay at its output by computing the maximum delays from its inputs and adding the module delay. So, we use new fresh variables for the inputs of a module being generated and assign the maximum delay to the output delay. This solution is depicted in the following function:

```
genCirDelay :: state -> delay -> circuit
genCirDelay (A, P, C, D) Dout = (i0, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (i1, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (i2, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (notGate B, (A, P, C, D)) <==
  domain [Dout] ((fd_min Dout) + notGateDelay) (fd_max Dout),
  genCirDelay (A, P, C, D) Dout == (B, (A, P, C, D))
genCirDelay (A, P, C, D) Dout = (andGate B1 B2, (A, P, C, D)) <==
  domain [Din1, Din2] ((fd_min Dout) + andGateDelay)(fd_max Dout),
  genCirDelay (A, P, C, D) Din1 == (B1, (A, P, C, D)),
  genCirDelay (A, P, C, D) Din2 == (B2, (A, P, C, D)),
  domain [Dout] (maximum (fd_min Din1)(fd_min Din2)) (fd_max Dout)
genCirDelay (A, P, C, D) Dout = (orGate B1 B2, (A, P, C, D)) <==
  domain [Din1, Din2] ((fd_min Dout) + orGateDelay) (fd_max Dout),
```

```
  genCirDelay (A, P, C, D) Din1 == (B1, (A, P, C, D)),
  genCirDelay (A, P, C, D) Din2 == (B2, (A, P, C, D)),
  domain [Dout] (maximum (fd_min Din1)(fd_min Din2)) (fd_max Dout)
```

Observing the rules for the AND and OR gates, we can see two new fresh domain variables for representing the delay in their inputs. These new variables are constrained to have the domain of the delay in the output but pruned with the delay of the corresponding gate. After the circuits connected to the inputs had been generated, the domain of the output delay is pruned with the maximum of the input module delays. Note that although the maximum is computed *after* the input modules had been generated, the information in the given output delay has been propagated to the input delay domains so that whenever an input delay domain becomes empty, the search branch is no longer searched and another alternative is tried. Putting together the constraints about area, power dissipation, cost, and delay is straightforward, since they are orthogonal factors that can be handled in the same way. In addition to the constraints shown, we can further constrain the circuit generation with other factors as fan-in, fan-out, and switching function enforcement, to name a few. Then, we could submit the following goal:

```
domain [A] 0 maxArea, domain [P] 0 maxPower,
domain [C] 0 maxCost, domain [D] 0 maxDelay,
genCir (A,P,C,D) == (B, S),  switchF B == sw
```

where `switchF` can be defined as the switching function that returns the result of a behavior `B` for all its input combinations, and `sw` is the function that returns the intended result (`sw` is referred as a problem parameter, as well as `maxArea`, `maxPower`, `maxCost`, and `maxDelay`).

```
data functionality = [bool]
switchF :: behavior -> functionality
switchF Behavior = [Out1,Out2,Out3,Out4,Out5,Out6,Out7,Out8] <==
      (Behavior false false false) == Out1,
      (Behavior false false true)  == Out2,
      (Behavior false true  false) == Out3,
      (Behavior false true true)   == Out4,
      (Behavior true  false false) == Out5,
      (Behavior true  false true)  == Out6,
      (Behavior true  true  false) == Out7,
      (Behavior true  true  true)  == Out8
```

Then, to generate a NOR circuit with `maxArea`, `maxPower`, `maxCost` and `maxDelay` equal 6, we could submit the following goal:

```
domain [A, P, C, D] 0  6, genCir (A,P,C,D) == (B, S),
switchF B == [true,false,false,false,false,false,false,false]
```

This goal has 24 possible answers, 4 of them are:

```
(1) ⟨true,{B == (notGate (orGate i0 (orGate i1 i2))), S == (_A,
_B,
     _C, _D)}, { },{_A ∈ 5..6, _B ∈ 5..6, _C ∈ 5..6, _D ∈ 5..6}⟩.
(2) ⟨true,{B == (notGate (orGate (orGate i2 i1) i0)), S == (_A,
_B,
     _C, _D)}, { }, {_A ∈ 5..6, _B ∈ 5..6, _C ∈ 5..6, _D ∈ 5..6}⟩.
(3) ⟨true,{B == (andGate (notGate i0) (notGate (orGate i1 i2))),
     S == (6, 6, _A, _B)}, { }, {_A ∈ 5..6, _B ∈ 4..6}⟩.
(4) ⟨true,{B == (andGate (notGate (orGate i2 i1)) (notGate i0)),
     S == (6, 6, _A, _B)}, { }, {_A ∈ 5..6, _B ∈ 4..6}⟩.
```

5 Related Work

[1] described an implementation of the FLP language Curry to enable the use of existing constraint solvers. As far as we know, our implementation is the first complete FLP system that includes truly solving on FD constraints although, recently, we have known about the existence of an (unpublished) implementation (called PAKCS) of the Curry language that supports (a small set of) FD constraints [3]. Specifically, PAKCS provides the following constraints: a set of arithmetic operations { $\#*, \#+, \ldots,$}, a membership constraint, an *all_different/1* constraint and an enumeration constraint that just provides naïve labeling.

Also, it is well-known that CLP(FD) is a successful declarative instance of CP and thus is strongly-related to our work. In fact, CLP(FD) is an instance of CFLP(FD) as any CLP(FD)-program can be straightforwardly translated into a CFLP(FD)-program. Observe that CFLP(FD) provides the main characteristics of CLP(FD), i.e., FD constraint solving, non-determinism, logical variables and relational form. Of course this determines initially a wide range of applications for our language. But CFLP(FD) is more than CLP(FD). Throughout the paper we have highlighted, by example, some CFLP(FD) features not existing in CLP(FD) . Particularly, Example 2 shows that CFLP(FD) provides functions, higher-order patterns, partial applications, combination of relational and functional notation, and types. This leads to an alternative way of expressing problems to that provided by CLP(FD) . Moreover, there are additional features existing in CFLP(FD) and not presented in CLP(FD) that have not been discussed so far as it is not the issue here and will be discussed in a further paper (currently under preparation). As an example we can cite the *lazy evaluation* of goals in which the arguments may be partially evaluated or evaluated just when they are necessary. Lazy evaluation opens new possibilities for FD constraint solving. For instance, CFLP(FD) enables the management of infinite lists of FD constraints.

Example 3. Consider the following function that generates an infinite list of FD variables constrained in the interval [0,N-1] for some integer N.

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

Consider also the polymorphic predefined function `take:: int -> [A] -> [A]` such that `take N L` returns a list containing the first `N` elements of the list `L`. Then the goal

```
take 3 (generateFD 10) == List
```

does not terminate with classical constraint solving as it tries to evaluate first the second argument yielding to an infinite list; however, a lazy evaluation generates just the first 3 elements of the list returning thus a correct answer.

In addition, our proposal can be considered in the context of multi-paradigm constraint programming, i.e., to combine CP with several paradigms in one setting. In this line, one decade ago, [10] presented an idea to combine characteristics of CLP, functional and concurrent languages and it was implemented in the language Oz. Despite Oz generalizes the CLP(FD) and concurrent constraint programming paradigms, it is very different to TOY(FD) as functional programming and constraints are not integrated. Moreover, instead of the typical LP approach of left-right first-depth, search strategies in Oz are encoded in *search procedures* to explore the search space.

There are also other declarative CP systems such as the algebraic CP languages (OPL [12], AMPL [5]). However, we think that our approach is far more declarative mainly since, first, those systems are not general-purpose programming languages, and, second, they do not benefit neither from complex terms and patterns nor from non-determinism.

6 Comparative Work

In this section we compare the performance of TOY(FD) with respect to related systems. One is PAKCS (cited in Section 5) that claims to be an efficient implementation. In the comparison we used the *Curry2Prolog* compiler, which is the most efficient implementation of Curry inside PAKCS. In addition, we also compared the performance of TOY(FD) with the FD constraint library of the efficient and well-known system SICStus Prolog (version 3.8.4).

Labeling. FD constraint solving can be seen as a combination of constraint propagation and labeling. Here, we consider two labelings, the naïve labeling (i.e., choose the leftmost variable of a list and then select the smallest value in its domain) and the *first fail* labeling (i.e., choose the variable with the smallest domain). The naïve labeling assures that both variable and value ordering are the same for all the systems and hence in many ways, although less efficient, is better for comparing the different systems when only one solution is required.

The Benchmarks. We have used a set of five classical benchmarks [11]: **send-more** (a cryptoarithmethic problem with 8 variables ranging over $\{0, \ldots, 9\}$), one linear equation and 36 disequations; **equation 10** and **equation 20** (systems of 10 and 20 linear equations respectively with 7 variables ranging over

Table 3. Performance Results for First Solution Search and Naïve Labeling.

Benchmark	*SICStus*	*TOY(FD)*	PAKCS	$\frac{TOY(FD)}{SICStus}$	$\frac{PAKCS}{SICStus}$	$\frac{PAKCS}{TOY(FD)}$
sendmore	10	10	40	1.00	4.00	4.00
equation10	20	70	80	3.50	4.00	1.14
equation20	30	130	160	4.33	5.33	1.23
queens (8)	10	20	30	2.00	3.00	1.50
queens (16)	1180	1220	4430	1.03	3.75	3.63
queens (20)	26430	31390	129510	1.18	4.90	4.12
queens (24)	57100	64770	326090	1.13	5.71	5.03
queens (30)	??	??	??	(?)	(?)	(?)
magic (64)	790	890	N	1.12	∞	∞
magic (100)	2270	2300	N	1.01	∞	∞
magic (150)	5840	5990	N	1.02	∞	∞
magic (200)	11450	11920	N	1.04	∞	∞
magic (300)	31280	34200	N	1.09	∞	∞

Table 4. Performance Results for First Solution Search and *First Fail* Labeling.

Benchmark	*SICStus*	*TOY(FD)*	$\frac{TOY(FD)}{SICStus}$	$\frac{SICStus(n)}{SICStus(f)}$	$\frac{TOY(FD)(n)}{TOY(FD)(f)}$
sendmore	5	5	1.00	2.00	2.00
equation10	10	50	5.00	2.00	1.40
equation20	20	110	5.50	1.50	1.18
queens (8)	10	15	1.50	1.00	1.33
queens (16)	40	50	1.25	29.50	24.40
queens (20)	80	160	2.00	330.37	196.18
queens (24)	70	90	1.28	815.71	719.66
queens (30)	130	660	5.07	∞	∞
magic (64)	320	330	1.03	2.46	2.69
magic (100)	640	690	1.07	3.54	3.33
magic (150)	1500	1510	1.00	3.89	3.96
magic (200)	2510	2620	1.04	4.56	4.54
magic (300)	6090	6180	1.01	5.13	5.53

$\{0, \ldots, 10\}$); **queens (N)** (place N queens on a $N \times N$ chessboard such that no queen attacks each other) and **magic sequences (N)** (calculate a sequence of N numbers such that each of them is the number of occurrences in the series of its position in the sequence).

Results. All the benchmarks were tested on the same SPARCstation under SunOs 5.8. Due to space limitations we only provide the results for first solution search. Table 3 shows the results using naïve labeling. The meaning for the columns is as follows. The first column gives the name of the benchmark used in the comparison. The next three columns show the running (elapsed) time (measured in milliseconds) to find the first answer for each system. The fourth and fifth columns indicate the slow-down of TOY(FD) and PACKS with

respect to SICStus. The last column shows the slow-down of the PAKCS with respect to our implementation.

Table 4 shows similar results but using first fail labeling. Observe that PAKCS is not included as it only provides naïve labeling (which is not very useful in practice as it is well-known). The meaning for the columns is as follows. The three first columns are as in Table 3. The fourth column indicates the slow-down of TOY(FD) with respect to SICStus. The last two columns show the slow-down of the solution using naïve labeling (n) with respect to the solution using first fail labeling (f).

In these tables, all numbers represent the average of a number of runs. The symbol ?? means that a solution was not received in a reasonable time and (?) indicates a non-determined value. The symbol **N** in the PAKCS column means that we could not formulate that benchmark because of insufficient provision for constraints. Particularly, the classical formulation of the magic sequence problem requires to use reified constraints in the form $X = Y \Leftrightarrow B$ with B being a (Boolean) FD variable. In these cases, when a problem cannot be expressed in PAKCS, the symbol ∞ is used in the average columns. All the benchmarks are available in [4].

7 Conclusions

We have presented CFLP(FD), a functional logic programming approach to FD constraint solving, which may be profitably applied to solve real-life problems. FD constraints are defined as functions and thus integrated naturally on FLP languages. Due to its functional component, CFLP(FD) provides better tools, when compared to CLP(FD), for a productive declarative programming. Due to the use of constraints, the expressivity and capabilities of this approach are clearly superior to those of both the functional and purely CP approaches.

We have described a formal language for CFLP(FD) and shown, by example, the benefits of integrating FLP and FD. For the execution mechanism of the language, we have seamlessly integrated constraint solving into a sophisticated, state-of-the-art execution mechanism for lazy narrowing. Our implementation, TOY(FD), translates CFLP(FD)-programs into Prolog-programs in a system equipped with a constraint solver. TOY(FD) provides a reasonably-complete set of FD constraints (including an acceptable number of practical options for labeling) and is fairly efficient as, in general, it is around two and five times faster than another CFLP(FD) implementation to come and also behaves closely to that of SICStus that means that the wrapping of SICStus by TOY does not increase significantly the computation time. The exception is in the solving of linear equations on which it is about three and five times slower. The reason seems to be in the process previous to the FD solver invocation that transforms the expressions in head normal form. This process produces an overhead when expressions (such as those for linear equations) involve a high number of arguments and sub-expressions.

We have also discussed briefly the advantages of CFLP(FD) wrt. CLP(FD). One of them is that CFLP(FD) enables to solve all the CLP(FD) applications as well as another problems closer to the functional setting. Moreover, the integration of FD constraints into the FLP paradigm provides extra advantages not existing in CLP(FD) such as types, higher-order computations, partial applications on constraints, functional notation and lazy evaluation among others.

In addition, we claim that our approach can be extended to other kind of interesting constraint systems, such as non-linear real constraints, constraints over sets, or Boolean constraints, to name a few.

References

1. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into prolog. In H. Kirchner and C. Ringeissen, editors, *3rd International Workshop on Frontiers of Combining Systems*, number 1794 in LNCS, pages 171–185. Springer-Verlag, 2000.
2. M. Carlsson, G. Ottosson, and B. Carlson. An pen-ended finite domain constraint solver. In U. Montanari and F. Rossi, editors, *9th International Symposium on Programming Languages: Implementations, Logics und Programs (PLILP'97)*, number 1292 in LNCS, pages 191–206, Southampton, UK, 1997. Springer-Verlag.
3. M. Hanus (editor). Pakcs 1.4.0, user manual. The Portland Aachen Kiel Curry System. Available from `http://www.informatik.uni-kiel.de/~pakcs/`, 2002.
4. A.J. Fernández, T. Hortalá-González, and F. Sáenz-Pérez. TOY(FD): User manual, latest Version. Available at `http://www.lcc.uma.es/~afdez/cflpfd/`, 2002.
5. R. Fourer, D.M. Gay, and B.W. Kernighan. Ampl: A modeling language for mathematical programming. Scientific Press, 1993.
6. J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. In Aart Middeldorp and Taisuke Sato, editors, *4th International Symposium on Functional und Logic Programming (FLOPS'99)*, number 1722 in LNCS, pages 1–20, Tsukuba, Japan, November 1999. Springer-Verlag. There is special issue of the Journal of Functional and Logic Programming, 2001. See `http://danae.uni-muenster.de/lehre/kuchen/JFLP`.
7. M. Hanus. The integration of functions into logic programming: A survey. *The Journal of Logic Programming*, 19–20:583–628, 1994. Special issue: Ten Years of Logic Programming.
8. J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19–20:503–581, 1994.
9. F.J. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques und Applications*, number 1631 in LNCS, pages 244–247, Trento, Italy, 1999. Springer-Verlag. The system and further documentation including programming examples is available at `http://babel.dacya.ucm.es/toy` and `http://titan.sip.ucm.es/toy`.
10. G. Smolka. The Oz programming model. In Jan Van Leeuwen, editor, *Computer Science Today*, number 1000 in LNCS, pages 324–343, Berlin, 1995. Springer-Verlag.

11. P. Van Hentenryck. *Constraint satisfaction in logic programming*. The MIT Press, Cambridge, MA, 1989.
12. P. Van Hentenryck. *The OPL optimization programming language*. The MIT Press, Cambridge, MA, 1999.
13. N-F. Zhou. Channel Routing with Constraint Logic Programming and Delay. In *9th International Conference on Industrial Applications of Artificial Intelligence*, pages 217–231. Gordon and Breach Science Publishers, 1996.