A Constraint Functional Logic Language over Finite Domains

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. We present a constraint functional logic programming language over finite domain for solving typical combinatorial problems. Our approach combines the power of constraint logic programming and the higher-order characteristics of functional logic programming (FLP).
Keywords: Constraint Programming, Functional Logic Programming, Finite Domains.

1 Introduction

Traditionally, Prolog has been a logic programming (LP) language used in many fields of artificial intelligence. However, it suffers a lack of expressiveness and also efficiency when solving combinatorial problems. Constraint programming languages add efficiency and expressiveness, but they lack several other important features as higher-order programming, functional applications, and lazy evaluation mechanisms. We provide a language combining features from logic, constraint, and functional programming (FP).

This paper describes our work of integrating FD constraints as functions in the FLP language TOY [5], which include pure LP and lazy FP programs as particular cases. Our work is a contribution for further augmenting the expressive power of FLP by adding the possibility of solving FD constraint problems in the functional logic setting. Our language gives rise to constraint functional logic programming on finite domain (CFLP(FD) for short) that seamlessly combines the power of the constraint logic programming over finite domains (CLP(FD) for short) with the higher-order characteristics of the FLP paradigm. Moreover, as far as we know, there is no concrete realization of a pure F(L)P language embodying FD constraints with reasonable efficiency. In this paper, we show a reasonably-efficient implementation of a CFLP(FD) language.

2 CFLP(FD) Programs

This section presents, by following the formalization given in [4], the basics about syntax, type discipline, and declarative semantics of CFLP(FD) programs.

* The work of this author has been partially supported by the project TIC2001-2705-C03-02 funded by the Spanish Ministry of Science and Technology.
** This work has been supported by the Spanish project PR 48/01-9901 funded by UCM.

2.1 CFLP(FD) Fundamental Concepts

Types and Signatures: We assume a countable set $\mathcal{TVar}$ 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

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

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 $\mathit{tvar}(\tau)$. 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 $\mathit{tvar}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\}$ 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 $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$, and we define the *arity* of $h \in DC^n \cup FS^n$ as $ar(h) = n$.

Table 1. Data Types for FD Constraints

```
data labelingType = 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 opRel = lt | eq | le | ge | gt | neq
data reasoning = value | domains | range
data options = on reasoning | complete bool
data typeprecedence = d (int,int,liftedInt)
data liftedInt = superior | lift int
data newOptions = precedences [typeprecedence] | path_consistency bool
                | static_sets bool | edge_finder bool | decomposition bool
```

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

- `int` $\to$ `int` $\to$ `int` to transform pairs of FD variables into FD variables (e.g., the arithmetic functions $\#+$, $\#-$, $\#*$ and $\#/$), or
- $\overline{\tau}_n \to$ `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

  $\mathit{Type}_{FD} = \{$`int`, [`int`], [`labelingType`], `statistics`, `opRel`, [`options`], [`newoptions`]$\}$.

  `int` is a predefined type for integers, and $[\tau]$ is the type 'list of τ'. The rest of data types in Type_{FD} are predefined types for FD and their definitions are

Table 2. The set of FD Constraints in TOY(FD)

RELATIONAL CONSTRAINTS	ARITHMETIC CONSTRAINTS (OPERATORS)
(#>) :: 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] → opRel → int → bool
(#\=) :: int → int → bool	scalar_product :: [int] → [int] → opRel → int → bool
COMBINATORIAL CONSTRAINTS	
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	count:: int → [int] → opRel → int → bool
element :: int → [int] → int → bool	cumulative :: [int] → [int] → [int] → int → bool
exactly :: int → [int] → int → bool	cumulative'::[int]→[int]→[int]→int→[newOptions]→bool
MEMBERSHIP CONSTRAINTS	
domain :: [int] → int → int → bool	
ENUMERATION CONSTRAINTS	STATISTICS CONSTRAINTS
labeling::[labelingType]→[int]→bool	fd_statistics :: statistics → int → bool
indomain :: int → bool	fd_statistics' :: bool

shown in Table 1. Examples of this sort of constraints are the relations $\#<$, and $\#>$ as well as the functions $all\_distinct'/3$, and $labeling/3$. In the following, $FS_{FD} \subset FS^n$ denotes the set of FD constraints that return a Boolean value.

The whole set of FD constraints supported in our language is shown in Table 2.

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* $e \in Exp_\perp$ have the syntax

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

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 $var(e)$.

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

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

where $X \in \mathcal{V}ar$, $c \in DC^k$, $0 \leq m \leq k$, $f \in FS^n$, $0 \leq m < n$ and $t_i \in Pat_\perp$ 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 [4]; 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 $1 \leq i \leq n$.

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

Finite Domains: A *finite domain* (FD) is a mapping $\delta : \mathcal{V}ar \to \wp(Integer)$ (as usual $\wp(C)$ denotes the powerset of the set C), with a unique extension $\hat{\delta} : Exp \to Exp$, which will be denoted also as δ, and *Integer* is the set of integers. We use $\delta = \{X_1 \in i_1, \ldots, X_n \in i_n\}$, which stands for the FD with domain $\{X_1, \ldots, X_n\}$ and satisfies $\delta(X_i) = i_i$ for all $1 \leq i \leq n$, where $i_i \subseteq Integer$.

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-typed CFLP(FD) Expressions and Programs

Well-typed Expressions: 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* $dom(T)$ and the *range* $ran(T)$ of a type environment are the set of all data variables (resp. type 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 $(h :: \tau) \in_{var} \Sigma_\bot$, $\sigma_t \in TSubst$.
AP $(\Sigma, T) \vdash_{WT} (e\ e_1) :: \tau$, if $(\Sigma, T) \vdash_{WT} e :: (\tau_1 \to \tau)$, $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1$, 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 $e \in Exp_\bot$ 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*.

Well-typed CFLP(FD) Programs: 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 $f :: \overline{\tau}_n \to \tau$ 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 *atomic 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 (included 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 $p\ t_1\ \ldots\ t_n = \mathit{true} \Leftarrow C_1, \ldots, C_k$

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

Here, we briefly describe part of TOY(FD), that is, our CFLP(FD) implementation that extends the TOY system [5] to deal with FD constraints. Table 2 shows the six different categories of FD constraints provided by TOY(FD). For reasons of space, we do not describe the constraints in detail and encourage the interested reader to visit the link proposed in [3] 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 provides an interface from TOY to the FD constraint solver of SICStus [2]. FD constraints are integrated in TOY(FD) as functions and evaluated internally by using mainly two predicates: $\mathit{hnf}(E, H)$, 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[3]:

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

(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) $\texttt{solve}(\texttt{L} \# \diamond \texttt{R}) \quad :- \texttt{hnf}(\texttt{L}, \texttt{L}'), \texttt{hnf}(\texttt{R}, \texttt{R}'), \{\texttt{L}' \# \diamond \texttt{R}'\}.$
where $\diamond \in \{<, <=, >, >=, =, \backslash=\}$.
(5) $\texttt{solve}(\texttt{C}\ \texttt{A}_1 \ldots \texttt{A}_n) :- \texttt{hnf}(\texttt{A}_1, \texttt{A}'_1), \ldots, \texttt{hnf}(\texttt{A}_n, \texttt{A}'_n), \{\texttt{C}(\texttt{A}'_1, \ldots, \texttt{A}'_n)\}.$
where C is any constraint returning a Boolean value.

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 idea is always the same: 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)

4.1 An Introductory TOY(FD) Example

Since CLP(FD) is an instance of CFLP(FD)(i.e., any CLP(FD)-program can be straightforwardly translated into a CFLP(FD)-program) our proposal determines initially a wide range of applications for our language. As an example below we show the TOY(FD) code to solve the classical arithmetic puzzle "send more money". TOY(FD) allows to use infix constraint operators such as #> to build the expression `X #> Y`, which is understood as `#> X Y`.

```
smm::int ->int ->int ->int ->int ->int ->int ->[labelingType] ->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 Scheduling Problem

Here, we consider a more realistic problem: the scheduling of tasks that require resources to complete, and have to fulfill precedence constraints. Figure 1 shows a precedence graph for six tasks which are labeled as tX^Y_{mZ}, where X stands for the identifier of a task t, Y for its time to complete, and Z for the identifier of a machine m (a resource needed for performing task tX).

The following program models the posed scheduling problem:

```
data taskName = t1 | t2 | t3 | t4 | t5 | t6
data resourceName = m1 | m2
type durationType = int
type startType = int
type precedencesType = [taskName]
type resourcesType = [resourceName]
```

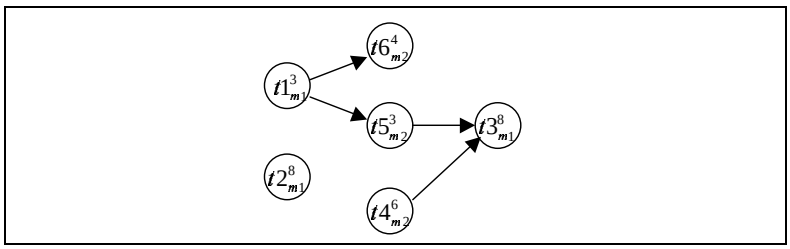

Fig. 1. Precedence Graph.

```
type task=(taskName,durationType,precedencesType,resourcesType,startType)

start :: task -> int
start (Name, Duration, Precedences, Resources, Start) = Start

duration :: task -> int
duration (Name, Duration, Precedences, Resources, Start) = Duration

schedule :: [task] -> int -> int -> bool
schedule TL Start End = true <== horizon TL Start End, scheduleTasks TL TL

horizon :: [task] -> int -> int -> bool
horizon [] S E = true
horizon [(N, D, P, R, S)|Ts] Start End :- domain [S] Start (End-D),
    horizon Ts Start End

scheduleTasks :: [task] -> [task] -> bool
scheduleTasks [] TL = true
scheduleTasks [(N, D, P, R, S)|Ts] TL:-precedeList (N, D, P, R, S) P TL,
    requireList (N, D, P, R, S) R TL, scheduleTasks Ts TL

precedeList :: task -> [taskName] -> [task] -> bool
precedeList T [] TL = true
precedeList T1 [TN|TNs] TL :- belongs (TN, D, P, R, S) TL,
    precedes (TN, D, P, R, S) T1, precedeList T1 TNs TL

precedes :: task -> task -> bool
precedes T1 T2 = (start T1) #+ (duration T1) #<= (start T2)

requireList :: task -> [resourceName] -> [task] -> bool
requireList T [] TL = true
requireList T [R|Rs] TL :- requires T R TL, requireList T Rs TL

requires :: task -> resourceName -> [task] -> bool
requires T R [] = true
requires (N1, D1, P1, R1, S1) R [(N2, D2, P2, R2, S2)|Ts] :-
    N1 /= N2, belongs R R2,
    noOverlaps (N1, D1, P1, R1, S1) (N2, D2, P2, R2, S2),
```

```
    requires (N1, D1, P1, R1, S1) R Ts
requires T1 R [T2|Ts] :- requires T1 R Ts

belongs :: A -> [A] -> bool
belongs R [] = false
belongs R [R|Rs] = true
belongs R [R1|Rs] = belongs R Rs

noOverlaps :: task -> task -> bool
noOverlaps T1 T2 :- precedes T1 T2
noOverlaps T1 T2 :- precedes T2 T1
```

A task is modeled (via the type `task`) as a 5-tuple which holds its name, duration, list of precedence tasks, list of required resources, and the start time. Two functions for accessing the start time and duration of a task are provided (`start` and `duration`, respectively) that are used by the function `precedes`. This last function imposes the precedence constraint between two tasks. The function `requireList` imposes the constraints for tasks requiring resources, i.e., if two different tasks require the same resource, they cannot overlap. The function `noOverlaps` states that two non overlapping tasks $t1$ and $t2$ either $t1$ precedes $t2$ or vice versa. The main function is `schedule` which takes three arguments: a list of tasks to be scheduled, the scheduling start time, and the maximum scheduling final time. These last two arguments represent the time window that has to fit the scheduling. The time window is imposed via domain pruning for each task's start time (a task cannot start at a time so that its duration makes its end time greater than the end time of the window; this is imposed with the function `horizon`). The function `scheduleTasks` imposes the precedence and requirement constraints for all of the tasks in the scheduling. Precedence constraints and requirement constraints are imposed by the functions `precedeList` and `requireList`, respectively.

With this model, we can submit the following goal, which defines the set of tasks, and asks for a possible scheduling in the time window (1,20):

```
Tasks == [(t1,3,[],[m1],S1), (t2,8,[],[m1],S2), (t3,8,[t4,t5],[m1],S3),
          (t4,6,[],[m2],S4), (t5,3,[t1],[m2],S5), (t6,4,[t1],[m2],S6)],
schedule Tasks 1 20, labeling [] [S1,S2,S3,S4,S5,S6]
```

4.3 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 [8]. 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. 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 with 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 *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.

```
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 2. 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 2).

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 3 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
```

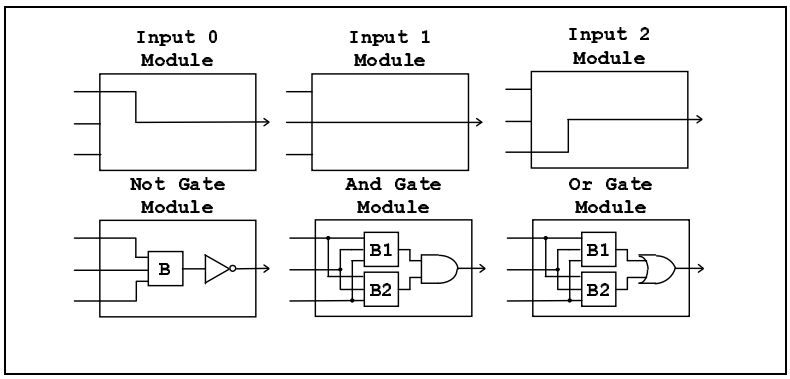

Fig. 2. Basic Modules.

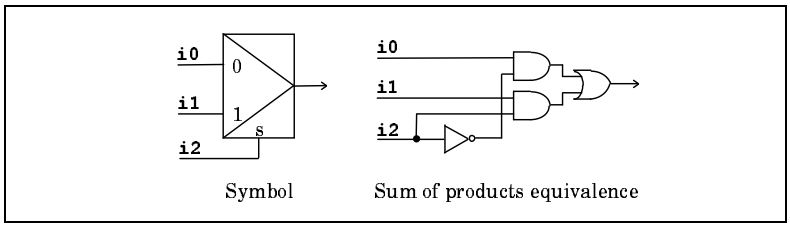

Fig. 3. Two-Input Multiplexer Circuit.

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[4]. As outcome, we get several solutions (⟨`i0`, {`P==0`}, {}, {}⟩, ⟨`i1`, {`P==0`}, {}, {}⟩, ⟨`i2`, {`P==0`}, {}, {}⟩, ⟨`not i0`, {`P==1`}, {}, {}⟩, ... , ⟨`not (not i0)`, {`P==2`}, {}, {}⟩, ... , which are denoted by a set of 4-tuples $\langle E, \sigma, C, \delta \rangle$ as a computed answer, where E is a TOY expression, σ is the set of variable substitutions, C is a

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

set of disequality constraints, and δ is the set of pruned domains). 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 successor 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)
power' (andGate C1 C2) = plus andGatePower (plus (power' C1) (power' C2))
power' (orGate C1 C2) = plus orGatePower (plus (power' C1) (power' C2))

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
```

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 no termination. FD constraints can be profitably applied to the representation of this problem as we show in the next example.

Example with 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 $\langle$area, power, cost, delay$\rangle$. 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 which is 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. (Please 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. Please 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),switchingFunction B == sw
```

where `switchingFunction` could be defined as the 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 refereed as a problem parameter, as well as `maxArea`, `maxPower`, `maxCost`, and `maxDelay`).

The solution to this problem has shown how to apply FD constraints to a functional logic language, which benefits from both worlds, i.e., taking functions, higher-order patterns, partial applications, non-determinism, logical variables, and types from FLP and domain variables, constraints, and propagators from the FD constraint programming. This leads to a more declarative way of expressing problems which cannot be reached from each counterpart alone.

5 Comparative Work

Previously to our implementation [1] described an efficient implementation of the FLP language Curry to enable the use of existing constraint solvers for LP. As far as we know, our implementation is the first complete FLP system that includes truly solving on FD constraints. However, 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 [6]. Specifically PAKCS provides the following constraints: (1) a set of arithmetic operations $\{\#*, \#+, \#-, \# = , \#/ =, \# <, \# <=, \# >, \# >=\}$, (2) a membership constraint similar to our

constraint *domain*/3, (3) an *all_different/1* constraint and (4) an enumeration constraint *labeling*/1 that just provides naïve labeling.

PAKCS is an efficient implementation that provides a smaller set of FD constraints than TOY(FD). We think it is worth to compare them. Due to space limitations we restrict this comparison to efficiency.

In the comparison we have used the *Curry2Prolog* compiler, which is the most efficient implementation of Curry inside PAKCS. In addition, we also compare the performance of our CFLP(FD) implementation with the FD constraint library of the efficient and well-known system SICStus Prolog (version 3.8.4).

Labeling It is well-known that 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 [7]: **sendmore** (a cryptoarithmethic problem with 8 variables ranging over $\{0,\ldots,9\}$), with one linear equation and 36 disequations; **equation 10** and **equation 20** (systems of 10 and 20 linear equations respectively with 7 variables ranging over $\{0,\ldots,10\}$); **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).

The programs **sendmore**, **equation 10** and **equation 20** test the efficiency of the systems to solve linear equation problems. The **N queens** and **magic sequences** programs are scalable and therefore useful to test how the systems works for bigger instances of the same problem. For fairness, we use exactly the same formulation of the problems for all systems as well as the same FD constraints.

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, the symbol ?? means that a solution was not received in a reasonable time and (?) indicates a vague value. The symbol **N** in the PAKCS column

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

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 [3].

In general, as it is expected, our implementation behaves closely to that of SICStus (which is known to be efficient) except for solving linear equations (in these cases it is about three and five times slower). The reason seems to be in the transformation process previous to the invocation of the FD solver. Expressions have to be transformed in head normal form what means that their arguments are also transformed in head normal form. Thus, there seems to be an overhead when expressions (such as those for linear equations) involve a high number of arguments and sub-expressions.

6 Conclusions

We have presented CFLP(FD), a functional logic programming approach to FD constraint solving and have described TOY(FD), a language for CFLP(FD) , whose implementation translates CFLP(FD)-programs into Prolog-programs in a system equipped with a constraint solver. We have also have shown that TOY(FD) is fairly efficient as, in general, behaves closely to that of SICStus FD solver.

In this paper we have not discussed in detail the benefits of integrating FLP and FD and have preferred to concentrate our efforts in showing the capabilities of TOY(FD) by means of programming examples. Of course, we believe that a detailed comparison of CFLP(FD) with respect to CLP(FD) is necessary. Unfortunately due to space limitations this is not done here and is the issue of a further paper (currently in preparation). For the moment, we can say that CFLP(FD) maintains the power and expressiveness of CLP(FD) whereas adds new characteristics not existing in CLP(FD) such as functional and curried notation, types, curried and higher-order functions (e.g., higher-order constraints), constraint composition, higher-order patterns, lazy evaluation and polymorphism among others.

Note: A revised version of this paper will be published in PADL'03.

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 open-ended finite domain constraint solver. In U. Montanari and F. Rossi, editors, *9th International Symposium on Programming Languages: Implementations, Logics and Programs (PLILP'97)*, number 1292 in LNCS, pages 191–206, Southampton, UK, 1997. Springer-Verlag.
3. 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.
4. 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 and 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`.
5. 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 and 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`.
6. 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.
7. P. Van Hentenryck. *Constraint satisfaction in logic programming.* The MIT Press, Cambridge, MA, 1989.
8. 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.