A Constraint Functional Logic Language for Solving Combinatorial Problems

Antonio J. Fernández-Leiva[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 approach over finite domain (CFLP(FD)) for solving typical combinatorial problems. Our approach adds to former approaches as (Constraint) Logic Programming, and Functional Logic Programming both expressiveness and further efficiency by combining combinatorial search with propagation. We integrate finite domain constraints into the functional logic language TOY. CFLP(FD) programs consist of TOY rules with finite domain constraints declared as functions. CFLP(FD) seamlessly combines the power of the constraint logic programming over finite domains with the higher order characteristics of the functional logic programming paradigm. This paper describes a language for CFLP(FD), the evaluation mechanism of CFLP(FD) programs, an implementation of our language for CFLP(FD) and programming examples that demonstrate the potential of the integration.
Keywords: Constraint Programming, Scheduling, Functional Logic Programming, Finite Domains.

1 Introduction

Traditionally, Prolog has been a logic programming 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.

Declarative programming (DP) is intended to separate the problem formulation from the procedure to solve the problem itself. Well known DP instances are logic programming (LP) on which the problem can be expressed in first order predicate calculus and functional programming (FP) that allows to express problems in terms of higher order functions. Recently, constraint logic programming (CLP) emerged to increase both the expressiveness and efficiency of LP programs [JM94]. The basic idea in CLP consists of replacing the classical LP unification by constraint solving on a given computation domain. Then, different instances of the computation domain generate different CLP instances that are used in the solving of problems of distinct nature.

Among the domains for CLP, the Finite Domain (FD) [Hen89] is one of the most and best studied since it is a suitable framework for solving discrete constraint satisfaction problems. The importance of the CLP languages based on FD is their impact in the industry since a lot of problems in the real life that involve variables ranging on discrete domains. This means that CLP languages for FD are appropriate to solve many real-world industrial problems. Unfortunately, literature lacks proposals to integrate FD constraints in the functional setting. This seems to be caused by the relational nature of the FD constraints that does not fit well in FP.

Another instance of DP is functional logic programming (FLP) that emerges with the aim to integrate the declarative techniques used in both FP and LP and that gives rise to new features not existing in FP or LP [Han94]. This paper describes our work of integrating FD constraints as functions in the FLP language TOY [LS99,Rod01], 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. 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 the integration of FD constraints into a FLP language. The implementation uses the efficient FD library provided by Sicstus Prolog.

* This work has been supported by the Spanish project PR 48/01-9901 funded by UCM.

Most of the work to integrate constraints in the DP paradigm has been developed on LP [CD96,CO+97]. However, there exist some attempts to integrate constraints in the functional (logic) framework. For instance, [AH+96,LS99] show how to integrate both linear constraints over real numbers and disequality constraints in the FLP language TOY. Also, [Lux01] describes the addition of linear constraints over real numbers to the FLP language Curry [Han00]. With respect to FD, the only functional system (to our knowledge) supporting FD constraints is Oz (currently called Mozart) [Smo95] which is a functional logic language based on concurrent constraint solving. However, this system is very different from pure FP systems as it is based on the concept of state over the object oriented paradigm. Also [AH00] provides a hint on how the integration of FD constraints in F(L)P could be carried out.

The structure of the paper is as follows: Section 2 shows our implementation of CFLP(FD), the TOY(FD) language. Section 3 introduces some program examples which show how to take advantage of the integration of FLP and FD. Finally, section 4 summarizes some conclusions and points out future work.

2 TOY(FD) : a CFLP(FD) Implementation

This section describes part of TOY(FD), that is, our CFLP(FD) implementation that extends the TOY system to deal with FD constraints and that also shows how to increase the FLP paradigm by integrating FD constraints as functions. (See [LS99] for a description of the base language.)

2.1 Constraints as Functions

TOY(FD) provides support for six different categories of FD constraints: (1) relational constraints, (2) arithmetic constraints, (3) combinatorial constraints, (4) membership constraints, (5) enumeration constraints and (6) statistics constraints.

Assume that L, L_1, L_2 are lists of integers or FD variables with length n; X, Y, N are FD variables or integer values; V, V_1, V_2 are integers and *RelOp* is a value of type *opRel* that represents a relational operator. Suppose also that $equiv(RelOp)$ is a function that returns the classical arithmetic operator equivalent to the value *RelOp* (i.e., $equiv(lt)$ is '#<', $equiv(eq)$ is '#=', $equiv(le)$ is '#<=', $equiv(ge)$ is '#>=', $equiv(gt)$ is '#>' and $equiv(neq)$ is '#\ =').

Relational Constraints include equality and disequality constraints in the form $e \# \diamond e'$ where $\diamond \in \{>, \geq, <, \leq, =, \neq\}$ and e and e' are integers, FD variables or functional expressions.

Arithmetic Constraints include all the classical arithmetic operators as well as the dedicated constraints 'sum/4' and 'scalar_products/5' where

- '$sum\ L\ RelOp\ V$' is true if
$$\sum_{e \in L} e\ \ equiv(RelOp)\ V$$
holds.
- '$scalar\_products\ L_1\ L_2\ RelOp\ V$' is true if the scalar product of L_1 and L_2 is related with the value V by the operator *RelOp*, i.e., if
$$L_1\ *_e\ L_2\ equiv(RelOp)\ V$$
is satisfied with $*_e$ defined as the usual scalar product of integer vectors.

Of course, the expressions constructed from both the arithmetic and relational constraints may be non-linear. The precedence of both the arithmetic and relational constraints are shown in Table 1.

Combinatorial Constraints include well known global constraints that are useful in the solving of problems formulated on discrete domains [Bel00]. TOY(FD) supports the following constraints:

- '$assignment/2$' is applied over two lists of domain variables of length n where each variable takes a value in $\{1, \ldots, n\}$ that is unique for that list. Then, '$assignment\ L_1\ L_2$' is true if for all $i, j \in \{1, \ldots, n\}$, and $X_i \in L_1$, $Y_j \in L_2$, then $X_i = j$ if and only if $Y_j = i$.

Table1. Priorities of Operators

RELATIONAL OPERATORS	ARITHMETIC OPERATORS
infix 30 #>, #<, #>=, #<=	infixr 90 #*
infix 20 # =, #\ =	infixl 90 #/
	infixl 50 #+, #-

- '*all_different* L' and '*all_distinct* L' are true if each variable in L is constrained to have a value that is unique among the list L and there is no duplicate integers in the list L, i.e., this is equivalent to say that for all $X, Y \in L$, $X \neq Y$. The difference between both constraints is that *all_different*/1 uses a complete algorithm that maintains the domain consistency [Reg94] whereas *all_distinct*/1 uses an incomplete one. There are extended versions that allow one more argument which is a list of options, where each option may have one of the following values
  1. 'on value', 'on domains' or 'on range' to specify that the constraint has to be woken up, respectively, when a variable becomes ground, when the domain associated to a variable changes or when a bound of the domain (in interval form) associated to a variable changes.
  2. 'complete true' or 'complete false' to specify if the propagation algorithm to apply is complete or incomplete.
- '*circuit* L_1' and '*circuit'* L_1 L_2' are true if the values in L_1 form a Hamiltonian circuit. This constraint can be thought of as constraining n nodes in a graph to form a Hamiltonian circuit where the nodes are numbered from 1 to n and the circuit starts in node 1, visits each node and returns to the origin. L_1 and L_2 are lists of FD variables or integers of length n, where the i-th element of L_1 (resp. L_2) is the successor (resp. predecessor) of i in the graph.
- '*element* X L Y' is true if the X-th element in the list L is Y (in the sense of FD).
- '*count* V L *RelOp* Y' is true if the number of elements of L that are equal to V is N and also N *equiv*(*RelOp*) Y.

Membership Constraints restrict variables to have values in a set of integers (i.e., an interval). The expression '*domain* L V_1 V_2' is true if each element in the list L belongs to the interval $[V_1, V_2]$.

Enumeration Constraints reactivate the search process when no more constraint propagation is possible. TOY(FD) provides the following constraints:

1. '*indomain* X' that assigns a value, from the minimum to the maximum in its domain, to X.
2. '*labeling Options* L' that is true if an assignment of the variables in L can be found such that all the constraints are satisfied. *Options* is a list of four elements of type 'labelingType' that allows to specify the nature of the search. Each element in this list may have a value in one of the following groups: (a) the first group controls the order in which variables are chosen for assignment (i.e., *variable ordering*) and allows to select the leftmost variable in L (`leftmost`), the variable with the smallest lower bound (`mini`), the variable with the greatest upper bound (`maxi`) or the variable with the smallest domain (`ff`). The value `ffc` extends the option `ff` by selecting the variable involved in the higher number of constraints. (b) The second group controls the *value ordering*, that is to say, the order in which values are chosen for assignment. For instance from the minimum to the maximum (`enum`), by selecting the minimum or maximum (`step`) or by dividing the domain in two choices by the midpoint (`bisect`). Also the domain of a variable can be explored in ascending order (`up`) or in descending order (`down`). (c) The third group demands to find all the solutions (`all`) or only one solution to maximize (resp. minimize) the domain of a variable X in L (`toMinimize X`) (resp. `toMaximize X`). (d) The fourth group controls the number of assumptions k (choices) made during the search (assumptions K).

3 Programming in TOY(FD)

Since CLP(FD) is an instance of CFLP(FD), any CLP(FD)-program can be straightforwardly translated into a CFLP(FD)-program, thus determining a wide range of applications for our language. We will not insist here on this matter, but prefer to concentrate on the extra capabilities of the language. We illustrate here different features of CFLP(FD) by means of example. We would like to emphasize that all the pieces of code are executable in TOY(FD) and the answers for example goals correspond to actual execution

of the program. Further programming examples in pure functional logic programming can be found in [LS99].

3.1 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 recourse needed for performing task tX).

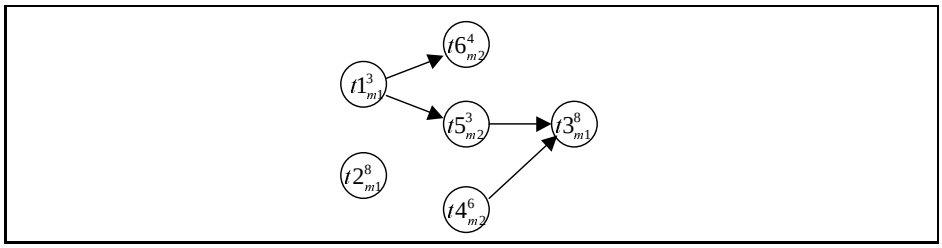

Figure1. Precedence Graph.

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]
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 T1 (TN, D, P, R, S),
    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 Rs,
    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 4-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, [t2], [m1], S1),
          (t2, 8, [], [m1], S2),
          (t3, 8, [t4,t5], [m1], S3),
          (t4, 6, [], [m2], S4),
          (t5, 3, [t1], [m2], S4),
          (t6, 4, [t1], [m2], S4)],
schedule Tasks 1 20, labeling [] [S1,S2,S3,S4,S5,S6]
```

3.2 A More Involved Example

A more interesting example comes from the hardware arena. 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 [Zho96].

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 it meets the switching function and it commits to the constraint physical factors.

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 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 *intensional* representation of functions. The alias `behavior` is used for representing the type `bool` $\rightarrow$ `bool` $\rightarrow$ `bool` $\rightarrow$ `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).

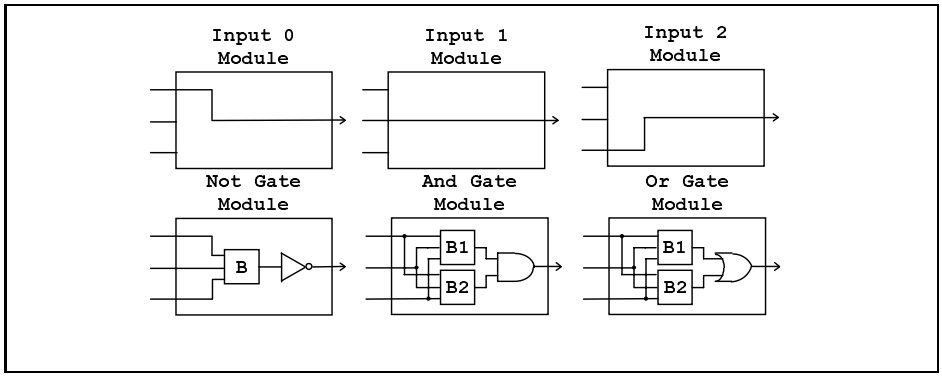

Figure2. Basic Modules.

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)
```

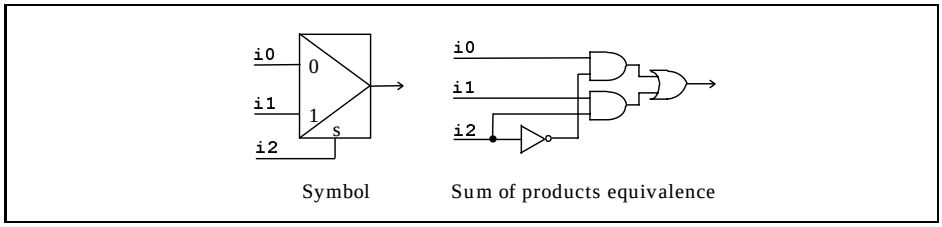

Figure3. Two-Input Multiplexer Circuit.

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
```

is evaluated to `true` and produces the substitution `P == 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 following goal (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 (Equivalently and more concisely, `power B < maxPower` could be submitted, but doing so we make the power unobservable.): `power B == P, P < maxPower`.

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 successor arithmetics, that is:

```
data nat = z | s nat

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

power' :: behavior -> nat
power' i0 = z
power' i1 = z
power' i2 = z
```

```
power' (notGate C) = s (power' C)
power' (andGate C1 C2) = s (plus (power' C1) (power' C2))
power' (orGate C1 C2) = s (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 no 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 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, type = 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 [A] 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. Note also that our approach is far more declarative than other constraint programming systems as algebraic constraint programming languages (OPL [Hen99], AMPL [FG+93]), mainly since they do not benefit neither from complex terms and patterns nor from non-determinism.

4 Conclusions

We have presented CFLP(FD), a functional logic programming approach to FD constraint solving, which we think may be profitably applied to solve typical problems in the artificial intelligence arena. We have

shown how FD constraints can be defined as functions and therefore 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 our approach are clearly superior to both those of the functional and purely constraint programming approaches. We have also presented the language TOY(FD) for CFLP(FD). Our proposal can be applied to a wide range of problems which include all CLP(FD) applications and typical uses of functional programming for combinatorial problems. In particular, we have 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 translates CFLP(FD)-programs into Prolog-programs in a system equipped with a constraint solver. 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

[AB92] A. Aggoun and N. Beldiceanu. *Extending CHIP to Solve Complex Scheduling and Placement Problems.* In Proc. of Journées Francophones de Programmation Logique, 1992.

[AH00] S. Antoy and M. Hanus. *Compiling Multi-Paradigm Declarative Programs into Prolog.* In Proc. of the 3rd International Workshop on Frontiers of Combining Systems, Springer LNCS 1794, pp. 171–185, Nancy, 2000.

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

[Bel00] N. Beldiceanu. *Global Constraints as Graph Properties on a Structured Network of Elementary Constraints of the Same Type.* In Proc. of 6th International Conference on Principles and Practice of Constraint Programming, Springer LNCS 1894, pp:52–66, Singapore, 2000.

[CD96] P. Codognet and D. Diaz. *Compiling Constraints in clp(FD).* The Journal of Logic Programming, 27(3):185–226, 1996.

[CO+97] M. Carlsson, G. Ottosson and B. Carlson. *An Open-Ended Finite Domain Constraint Solver.* In Proc. of 9th International Symposium on Programming Languages: Implementations, Logics and Programs, Springer LNCS 1292, pp:191–206, Southampton, 1997.

[FG+93] R. Fourer, D.M. Gay and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Scientific Press, 1993.

[GH+01] J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming.* FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. See `http://danae.uni-muenster.de/lehre/kuchen/JFLP`

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

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

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

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

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

[LS99] F.J. López-Fraguas and J. Sánchez-Hernández. *TOY: A Multiparadigm Declarative System.* In Proc. of the 10th International Conference on Rewriting Techniques and Applications, Springer LNCS 1631, pp. 244–247, Trento,, 1999. The system and further documentation including programming examples is available at `http://titan.sip.ucm.es/toy`

[Lux01] W. Lux. *Adding Linear Constraints over Real Numbers to Curry.* In Proc. of 5th International Symposium on Functional and Logic Programming, Springer LNCS 2024, pp. 185–200, Tokyo, 2001.

[Reg94] J-C. Régin. *A Filtering Algorithm for Constraints of Difference in CSPs.* In Proc. of 12th National Conference on Artificial Intelligence, vol. 1, pp: 362–367, AAAI Press, 1994.

[Rod01] M. Rodríguez-Artalejo. *Functional and Constraint Logic Programming.* In Constraints in Computational Logics, Springer LNCS 2002, pp. 202–270, 2001.

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

[Zho96] N.F. Zhou, *Channel Routing with Constraint Logic Programming and Delay.* In Proc. of the 9th International Conference on Industrial Applications of Artificial Intelligence, pp. 217-231, Gordon and Breach Science Publishers, 1996.