

# Declarative Programming with Real Constraints<sup>1</sup>

T. Hortalá-González, F. J. López-Fraguas,  
J. Sánchez-Hernández, E. Ullán-Hernández

TECHNICAL REPORT SIP - 5997

Dep. Sistemas Informáticos y Programación  
Univ. Complutense de Madrid

APRIL 97

---

<sup>1</sup>The authors have been partially supported by the Spanish CICYT (project TIC 95-0433-C03-01 'CPD') and the ESPRIT Working Group 22457 (CCL-II).

# Declarative Programming with Real Constraints

T. Hortalá-González, F. J. López-Fraguas,  
J. Sánchez-Hernández, E. Ullán-Hernández

*Universidad Complutense de Madrid, Dpto. de Informática y Automática  
Facultad de CC. Matemáticas, Avda. Complutense s/n, 28040 Madrid, Spain  
email:{alp94teja,fraguas,evah}@eucmax.sim.ucm.es, jaimew@eucmos.sim.ucm.es*

**Abstract.** We present a declarative language –  $CFLP(\mathcal{R})$  – which enhances functional logic programming with constraint solving capabilities.  $CFLP(\mathcal{R})$  features: polymorphic types, nondeterministic functions, lazy evaluation, higher order (even logic) computations, arithmetical constraints over real numbers and disequality constraints over syntactic terms. The features of the language are shown by means of examples, attempting to demonstrate the interest of  $CFLP(\mathcal{R})$ . The execution mechanism of the language results of a simple combination of lazy narrowing (with a sophisticated strategy, as realized in up to date functional logic languages) and constraint solving. The language has been implemented by means of translation of source programs into a Prolog system supporting real constraint solving. This shows the practicability of the proposal.

## 1 Introduction

Constraints play a central role in present days research, development and application of logic programming (LP) languages (see [15] for a survey). Most of the interest in this field started with the proposal of the  $CLP(\mathcal{X})$  scheme [14], a general framework for constraint logic programming (CLP) languages. The  $CLP(\mathcal{X})$  scheme was conceived hand by hand with one of its most prominent instances, the language  $CLP(\mathcal{R})$  [16], which extended traditional LP by the use of real arithmetic constraints for expressing conditions in clauses and for describing solutions, and combined SLD-resolution with a mechanism for solving linear constraints. Due to the variety of its applications and to the clarity of its conception,  $CLP(\mathcal{R})$  has had a great influence in later CLP languages. We think that the two main merits of CLP have been to convert domain dependent computations (as arithmetic calculations, in the case of  $CLP(\mathcal{R})$ ), which are impure (but essential to practice) features of logic programs, into declarative ones, and to convert constraint programming (a technology independently developed) into a very rich programming paradigm.

Traditional LP and CLP lack features which are recognized as powerful tools for a productive declarative programming: polymorphic type discipline, lazy evaluation, higher order (HO) features. Some effort has been devoted to those matters in the LP community (see e.g. [25, 23, 24]), but nevertheless it seems that those issues are still better supported in functional programming (FP). This is one of the main reasons advocated for the the integration of the FP and LP paradigms, which constitutes another important branch in the evolution of declarative languages (see [10] for a survey).

With functional logic programming (FLP) a problem arises: while FP smoothly support domain dependent computations in a declarative way, this property is lost in the usual approach to FLP, which is based in some Herbrand-like representation of data. In particular the problem of nondeclarativeness of arithmetic re-appears. To incorporate constraints to FLP appears then as a natural, not merely speculative, interesting task. In [18, 19] a theoretical general scheme –  $CFLP(\mathcal{X})$  – for *constraint functional logic programming* (CFLP) was proposed, with the aim of extending lazy FLP in the same way that  $CLP(\mathcal{X})$  extended traditional LP. There are other proposals for CFLP [22, 4, 20] but, as far as we know, they have not fructified in the development of concrete, practical languages.

In this paper we describe the language  $CFLP(\mathcal{R})$  which incorporates real arithmetic constraints to a higher order lazy nondeterministic functional logic language in the spirit of [7, 8]. The language incorporates also syntactic disequality constraints in the way of [2]. The language has been implemented in the system  $TOY(\mathcal{R})$  (<http://mozart.mat.ucm.es/incoming/comprimidos/toyr.tar.gz>), which is an extension of the FLP system  $TOY$  (<http://mozart.mat.ucm.es/incoming/comprimidos/toy.tar.gz>). The rest of the paper is organized as follows: in Sect. 2 we shortly describe the language and its syntax. Section 3 contains a non trivial example showing some of the possibilities of our language. In Sect. 4 we informally explain the execution mechanism of the language, and sketch how to translate  $CFLP(\mathcal{R})$ -programs into a constraint logic programming language. Sect. 5 includes some words about the actual implementation. Finally, Sect. 6 summarizes some conclusions.

## 2 Description of the language

We give here quite a succinct account of the constructs of the language, which will be exemplified in Sect. 3.  $CFLP(\mathcal{R})$  programs consists of *datatype*, *type alias* and *infix operator* definitions, rules for defining *functions* and clauses for defining *predicates*. Syntax is most borrowed from Haskell [11] (the main exception being that variables begin with upper-case letters whereas constructors use lower-case). Note, in particular, that functions are *curried* and the usual conventions about associativity of application hold.

*Datatype definitions* like `data nat = zero | suc nat`, define new (possibly polymorphic) *constructed types* and determine a set of *data constructors* for each type. The set of all data constructor symbols will be noted as  $CS$  ( $CS^n$  for all constructors of arity  $n$ ).

*Types*  $\tau, \tau', \dots$  can be constructed types, tuples  $(\tau_1, \dots, \tau_n)$ , or functional types of the form  $\tau \rightarrow \tau'$ . As usual,  $\rightarrow$  associates to the right.  $CFLP(\mathcal{R})$  provides predefined types such as  $[A]$  (the type of polymorphic lists, for which Prolog notation is used), *bool* and *real*. Furthermore, *type alias* definitions like `type point=(real,real)` are allowed. Type alias are simply macros, but they are useful for writing more abstract, self-documenting programs. Type classes are not considered in  $CFLP(\mathcal{R})$ .

Any  $CFLP(\mathcal{R})$ -*program* has an associated set  $FS$  of function symbols, each with a corresponding *program arity*. We note by  $FS^n$  the set of function symbols of program arity  $n$ . Some of the functions are primitive, which include the arithmetic functions  $+$ ,  $-$ ,  $*$ ,  $/$ . Each function  $f \in FS^n$  has an associated principal type of the form  $\tau_1 \rightarrow \dots \rightarrow \tau_m \rightarrow \tau$  (where  $\tau$  does not contain  $\rightarrow$ ).  $m$  is called the *type arity* of  $f$ , and must be  $n \leq m$ . As usual in functional programming, types are inferred and, optionally, can be declared in the program. Each non primitive  $f \in FS^n$  is defined by a set of rules of the form  $f t_1 \dots t_n = e \Leftarrow \varphi_1, \dots, \varphi_k$ , where  $(t_1 \dots t_n)$  is a tuple of linear (i.e., with no repeated variable) *patterns* (see below),  $e$  is an *expression* (see below), and each *constraint*  $\varphi_i$  has the

form  $e \diamond e'$ , where  $\diamond \in \{=, /, <, >, \leq, \geq\}$ . A rule has a conditional reading:  $f t_1 \dots t_n$  can be reduced to  $e$  if the constraint  $\varphi_1, \dots, \varphi_k$  is satisfied. The constraint  $\Leftarrow \varphi_1, \dots, \varphi_k$  is omitted if  $k = 0$ . In  $e = e'$  or  $e / e'$ ,  $e$  and  $e'$  must have the same type.

A *pattern*  $t$  is defined as  $t ::= X \mid r \mid (t_1, \dots, t_n) \mid c t_1 \dots t_n \mid f t_1 \dots t_n$ , where  $r$  is a real number,  $c \in CS^m$ ,  $n \leq m$ ,  $f \in FS^m$ ,  $n < m$ . One distinguished feature of our patterns is that partial application of  $c$  and  $f$  (which have functional type) are allowed within them, which are then called *HO patterns*. This corresponds to an *intensional* point of view of HO patterns (see [6, 8]). Observe that in this approach function symbols, when partially applied, behave as data constructors.

*Expressions*  $e$  are of the form  $e ::= X \mid r \mid (e_1, \dots, e_n) \mid c e_1 \dots e_n \mid f e_1 \dots e_n \mid e e_1 \dots e_n$ , where  $c \in CS^m$ ,  $n \leq m$ ,  $f \in FS^m$ ,  $n \leq m$ . Of course expressions are assumed to be well-typed.

In addition to the arithmetic functions, we have the boolean primitive functions  $=, /, <, \leq, >, \geq$  (notice the overloading of symbols, which can be solved by the context), which can be understood as defined by the rules

$$X = Y = \text{true} \iff X = Y \qquad X = Y = \text{false} \iff X \neq Y$$

and similarly for the others. For *if\_then\_else* (which can be defined by rules), the usual syntax *if \_then \_else \_* is allowed.

Another distinguished feature of our language is that functions can be *nondeterministic* (no confluence properties for the program are required). For example, the rules  $X // Y = X$  and  $X // Y = Y$  constitute a valid definition for a ‘choice function’  $//$ <sup>1</sup>. The reduction of  $(0//1)$  produces first 0 and then, if backtracking is required, 1. Following [7, 8] our language adopts the so called *call-time choice* semantics for nondeterministic functions [13].

Predicates are particular cases of boolean functions, for which clausal notation is allowed. The syntax  $p t_1 \dots t_n :- \varphi_1, \dots, \varphi_k$  is a sugaring for  $p t_1 \dots t_n = \text{true} \Leftarrow \varphi_1, \dots, \varphi_k$ . Some more sugaring:  $\Leftarrow \dots, b, \dots$  is an abbreviation for  $\Leftarrow \dots, b = \text{true}, \dots$ . With these sugarings in mind and some obvious changes (like currying elimination), it is easy to see that (pure)  $CLP(\mathcal{R})$ -programs are  $CFLP(\mathcal{R})$ -programs.

If  $e, e'$  have type real, then in order to satisfy a constraint  $e \diamond e'$ ,  $e$  and  $e'$  must be reduced to primitive expressions (i.e., involving no defined function)  $r, s$  such that  $r \diamond s$  is compatible with the accumulated so far arithmetical constraints. Compatibility is checked by a constraint solver. We stick here to the  $CLP(\mathcal{R})$  philosophy: non-linear constraints are delayed. For the non real case, ‘=’ and ‘/’ are interpreted as strict equality and disequality. Such constraints are transformed into some kind of solved forms. For instance,  $X \neq [Y, Z]$  (which covers infinitely many positive cases) is a solved form (see [2, 19] for details).

*Goals* take the form  $\varphi_1, \dots, \varphi_k$ . Solving a goal means obtaining conditions (a mixture of substitution and constraints) over their variables ensuring the satisfiability of the goal.

To give a complete semantic characterization of  $CFLP(\mathcal{R})$  is far out of the scope of this paper. We may cite some works giving solid foundations to different aspects of the language: [19] explains how the untyped FO subset of  $CFLP(\mathcal{R})$  can be characterized as an instance of the  $CFLP(\mathcal{X})$  scheme [18, 19]. [2, 19] study disequality constraints. [7, 8] investigate nondeterministic functions in an FLP setting, for FO and HO cases respectively. Polymorphic (and algebraic) types in an FLP setting are treated in [3].

---

<sup>1</sup>// is an example of *infix operator*, which are like in Haskell. Sections (partial applications of infix operators) are also allowed.

### 3 Programming in $CFLP(\mathcal{R})$

We illustrate here different features of  $CFLP(\mathcal{R})$  by means of an example. We would like to emphasize that all the pieces of code (for which we use `typewriter` style) are executable in  $TOY(\mathcal{R})$  as they are (except for some reformatting, due to space reasons) and that answers for example goals correspond to real execution of the program. We sometimes omit the definition of (mainly HO) functions which are ‘standard’ in functional programming.

**Types, functions, HO functions.** Our program example deals with regions (sets of points) in the plane. Following a usual approach in functional programming, we identify regions with their characteristic functions. Points in the plane are simply pairs of real numbers. So we introduce the type alias declarations `type point = (real,real)` and `type region = point -> bool`. In order to give more ‘set-flavour’ to regions, we declare an operator `infixr 50 <<-` (read ‘belongs to’ or ‘is in’) and define `P <<- R = R P`.

Up to now, all definitions are almost pure sugaring, but nevertheless they contribute to the writing of a more readable code. Next, we define a variety of regions or region generators: the empty region, the whole plane, regions consisting of a single point, rectangles (defined by their left-down and right-up corners), circles (defined by their center and radius). The infix operators `infixr 40 /\` and `infixr 30 \/` are used for boolean (sequential) conjunction and disjunction, whose definitions are standard. `not` is boolean negation.

```
emptyReg, thePlane:: region          point:: point -> region
emptyReg P = false                    point P Q = P==Q
thePlane P = true

rectangle:: point -> point -> region
rectangle (A,B) (C,D) (X,Y) = A <= X /\ X <= C /\ B <= Y /\ Y <= D

circle:: point -> real -> region
circle (A,B) R (X,Y) = (X-A)*(X-A)+(Y-B)*(Y-B) <= R*R

false /\ Y = false          true  \/ Y = true          not true = false
true  /\ Y = Y              false \/ Y = Y            not false = true
```

Observe that all these regions are two-valued boolean functions. We define now some basic operations with regions: intersection, union, complement.

```
intersct, union:: region -> region -> region
outside:: region -> region

intersct R R' P = P <<- R /\ P <<- R'          outside R P = not (P <<- R)
union    R R' P = P <<- R \/ P <<- R'
```

Notice that all these are HO functions, since regions are functions. The HO technology can be used for readily defining presumably useful functions like the following for intersecting or unioning lists of regions. The standard (in FP) HO function `foldr` is used.

```
intersctAll, unionAll:: [region] -> region
intersctAll = foldr intersct thePlane      unionAll = foldr union emptyReg
```

We can define many other operations on regions, like movements. We introduce first vectors in the plane (complex numbers) and arithmetic operations (`+. , -. , *. , /.`) over them, whose straightforward definitions are omitted. We introduce a new type-alias for vectors `type vector = (real,real)`. Now it is easy to define, e.g, translations (determined by a vector) or homotheties (determined by the center and the factor of scale).

```

translate:: vector -> region -> region
homothety:: point -> real -> region -> region
translate V R P = (P -. V) <<- R
homothety C F R P = (C +. (P -. C) /. (F,0)) <<- R

```

**Lazy evaluation, infinite structures.** Since the language uses lazy evaluation, we can define infinite structures like the infinite lists of regions obtained by application of `rayItems` which are then unioned to form an ‘infinite region’ in `ray`.

```

rayItems:: vector -> region -> [region]
ray:: vector -> region -> region
rayItems V R = [R|rayItems V (translate V R)]
ray V R = unionAll (rayItems V R)

```

As a subtlety, observe that `ray V R` can give only the value `true` (if a point `P` is not in `ray V R` then the computation never ends), therefore it computes a semicharacteristic function. This is not really our fault, since infinite unions of recursive sets are in general not recursive, but only recursively enumerable.

**Reversibility of functions.** If we disregard syntactical differences, the program written so far works as a functional (e.g. Haskell) program. The Haskell evaluation of an expression  $e$  would correspond to solving the goal  $X == e$ . But our language provides a much more flexible use of functions, due to the incorporation of constraint solving. For instance, the goals `P <<- rectangle (0,0) (1,1)` and `(1,1) <<- outside (point P)` produce the respective answers `P == (_A, _B) { _B>=0, _B<1, _A>=0, _A<1 }` and `P /= (1, 1)`. Such goals make no sense in a functional language. In standard functional logic languages they would fail (or produce a runtime error) since real arithmetic is an impure feature which is not reversible. Notice that the disequality in the last answer is not an arithmetic constraint but a syntactic disequality constraint, one of the abilities of our language.

As a consequence of the possibility of solving goals with logical variables ranging over reals, we can define functions in  $CFLP(\mathcal{R})$  which are not Haskell-like functions anymore. For instance, we can define the predicate

```

move:: vector -> region -> region
move (U,V) R P :- 0 <= A, A <= 1, translate (A*U,A*V) R P

```

which recognizes if the region traversed by the translation of a region `R` through a vector `(U,V)` meets the point `P`. Notice the existential reading of the variable `A` in the condition<sup>2</sup>, whose appearance precludes a direct counterpart of this definition in a functional language.

In  $CFLP(\mathcal{R})$ , the goal `P <<- move (1,2) (rectangle (0,0) (1,1))` gives the answer `P == (_A,_B) { _A-0.5*_B<1, _A-0.5*_B>= -0.5, _A>=0, A<2, _B>=0, _B<3}`.

**Comparing  $CFLP(\mathcal{R})$  with  $CLP(\mathcal{R})$ : a discussion.** It is generally accepted that polymorphic types, lazy evaluation and HO features are powerful tools for writing concise, highly descriptive and well structured programs. This is justification enough for a functional extension of a constraint logic language, but there are further reasons, not so

---

<sup>2</sup>This existential variable is responsible of having defined `move` as a predicate that, as in the case of `ray`, computes a semicharacteristic function. Again this is not our fault, since existential quantification does not preserve recursivity.

frequently discussed, which can be well illustrated by our program example.

The definition of, e.g., rectangles is much more cumbersome in a CLP language, if both values `true` and `false` are expected as possible when asking if a point is in a rectangle. The obvious CLP-definition

```
rectangle((A,B),(C,D),(X,Y)) :- {A=<X,X=<C,B=<Y,Y=<D}.
```

(`{ _ }` in CLP-code denotes invocation to the constraint solver) only serves for obtaining positive answers. As it is well known, negation as failure can only deal correctly with ground goals, and constructive negation is very hard to implement.

Of course, it is possible to redefine `rectangle` adding an argument with possible values `true` and `false`. The new CLP-definition could look like this:

```
rectangle((A,B),(C,D),(X,Y),true) :- {A =< X, X =< C , B =< Y , Y =< D}.
rectangle((A,B),(C,D),(X,Y),false) :- {A > X ; X > C ; B > Y ; Y > D}.
```

This is less clear than the *CFLP*( $\mathcal{R}$ ) definition and, in addition, the CLP-goal `rectangle((0,0),(1,1),P,false)` gives answers with some amount of redundancy, in contrast to the analogous *CFLP*( $\mathcal{R}$ )-goal `false == P <<- rectangle(0,0)(1,1)` which gives the incompatible answers

```
P == (_A, _B)    P == (_A, _B)    P == (_A, _B)    P == (_A, _B)}
{ _B>1.0 }      { _B<0.0 }      { _A>1.0 }      { _A<0.0 }
{ _A>=0.0 }     { _A>=0.0 }
{ _A=<1.0 }     { _A=<1.0 }
```

To obtain incompatible answers we can explicitly incorporate to each disjunct in the second clause for *rectangle/4* the negation of the previous ones, resulting in a much more complicated and error-prone way of proceeding. Trying to be more ‘abstract’, we could recognize the usefulness of programming (the relational counterpart of) boolean functions for `=<`, `>=`, ... and for the boolean connectives *and*, *or*, ... We could then write

```
leq(X,Y,true)  :- {X =< Y}.           and(false,X,false).
leq(X,Y,false) :- {X >= Y}.          and(true,X,X).
```

and similarly for the others. We could rewrite *rectangle/4* into the more ‘functional’ version

```
rectangle((A,B),(C,D),(X,Y),Truth) :-
  leq(A,X,T1),leq(X,C,T2),leq(B,Y,T3),leq(Y,D,T4),
  and(T1,T2,U),and(U,T3,V),and(V,T4,Truth).
```

In fact, this resembles closely our *CFLP*( $\mathcal{R}$ ) definition of rectangles, but is nevertheless much less readable due to the difficulty in this case of using infix operators (since predicates are ternary) and, more significantly, to the need of unnesting function applications (of `and` in this case). Furthermore, this new definition again results in non incompatible answers for many goals. But this is not all. The flattening of nested applications has the unpleasant consequence that much computational effort can be wasted if we are mimicing with predicates (like `and/3`) non strict functions (like `\&`, which is not strict in its second argument). For instance, for solving the CLP-goal `rectangle((0,0),(2,2),(-1,1),false)`, all the conditions in the body of the last clause must be proved, while it seems that after proving `leq(0,-1,false)`, we can immediately conclude that the conjunction is false. Another unpleasant effect of the same fact is that search spaces can be increased. For instance, the

CLP-goal `rectangle((0,0),(2,2),(-1,P),false)` produces three answers constraining `P`, instead of succeeding without any condition over `P`, as happens with the  $CFLP(\mathcal{R})$ -goal `false == (-1,P) <<- rectangle(0,0)(2,2)`. Of course, CLP-programmers can argue that the above definition of `rectangle` is still a bad one, and that intermediate truth values should be passed through invocations to `leq`, avoiding the comparisons `leq(,-,-)` when a value `false` is obtained. This results in something similar to the following CLP-definition.

```
rectangle((A,B),(C,D),(X,Y),Truth) :-
    leq(A,X,T1),leqIfTrue(T1,X,C,T2),leqIfTrue(T2,B,Y,T3),leqIfTrue(Y,D,T4).

leqIfTrue(false,_,_,false).      leqIfTrue(true,X,Y,T) :- leq(X,Y,T).
```

We think that the  $CFLP(\mathcal{R})$  formulation is clearly simpler, more ‘descriptive’, and better structured than this one.

**Higher order logic computations.** In  $CFLP(\mathcal{R})$ , variables with HO type are allowed during execution. In this case, the system tries to instantiate the variable to a HO pattern which satisfies the goal. For instance, for the goal `(0,0) <<- R`, we expect `R` to be instantiated to HO patterns denoting regions containing the point `(0,0)`. Some (selected) answers provided by the system are:

```
R == (point (0, 0))
R == (rectangle (_A, _B) (_C, _D)) { _C>=0, _A<=0, _B<=0, _D>=0 }
R == (outside emptyReg)
R == (outside (point _A)) { _A /= (0, 0) }
R == (outside (outside (point (0, 0))))
```

**Higher order patterns.** Among the HO capabilities of our language, it is remarkable the possibility of using HO patterns in left hand sides of rules for function definitions. In practice this means that we can distinguish cases, when defining a HO function, according to different ‘intensional shapes’ that an argument (of HO type) can adopt.

Let us revise, for instance, our definition of `intersct`. It was a good definition, but we may want to take into account the fact that intersecting the empty region with any other gives the empty region, or that by intersecting rectangles we obtain again a rectangle. HO patterns allow us to express easily this kind of things, as it is done in the following variation of `intersct`. A similar treatment could be done for union of regions.

```
intersct':: region -> region -> region
intersct' emptyReg R = emptyReg
intersct' R emptyReg = emptyReg
intersct' (rectangle (A,B) (C,D)) (rectangle (A',B') (C',D')) =
    if (A'' <= C'') /\ (B'' <= D'') then rectangle (A'',B'') (C'',D'')
    else emptyReg
    <== A''== max A A' , B''== max B B' , C''== min C C' , D''== min D D'
% Now, the default rule
intersct' R R' = intersct R R' % Uses the old definition
<== % if none of the above situations apply
R /= emptyReg, R' /= emptyReg,
(R,R') /= (anyRectangle,anyRectangle).
```

```

anyRectangle = rectangle undefined undefined
undefined = undefined <== true == false

```

The last rule of `intersect'` requires some explanations. If we had written no condition, then the default rule would be applicable to any pair of regions, including the empty region and rectangles. It is not incorrect, but will produce two alternative computations for such cases, while the idea was: proceed in a particular way for the the empty region and rectangles, and in a default way in the rest of the cases. The condition of the last rule is interesting by itself, since it reveals a somehow surprising aspect of our syntactic disequality constraints, as is the capability of expressing certain universally quantified disequations. We explain this now. The (constant) function `undefined`, whose evaluation simply fails, can be thought as denoting the least defined element ( $\perp$ ) of each type. The conditions  $t == \perp$  and  $t /= \perp$  do not hold, for any  $t$  (including  $\perp$  itself). As a consequence, a condition of the form  $X /= c$  `undefined` is equivalent to say  $\forall Y (X /= c Y)$ , i.e.,  $X$  does not take the form  $(c \_)$ . In our example, the condition  $(R, R') /= (\text{anyRectangle}, \text{anyRectangle})$  expresses that either  $R$  or  $R'$  does not take the form  $(\text{rectangle } \_)$ , so the rule for intersecting rectangles cannot be applied.

HO patterns can be used for partially defining functions which are computable in particular cases, but not in general. As an example, consider inclusion of regions, which is undecidable in general. But it is possible to define inclusion for particular cases. We can make explicit such cases or, as we do here, reuse `intersect'` for capturing them. The case of 'non-inclusion' can be treated with more generality.

```

(<<):: region -> region -> bool
R << R' = true  <== intersect' R R' == R
R << R' = false <== P <<- R, not (P <<- R')

```

**Nondeterministic functions** Nondeterministic (ND) functions can be profitably used instead of relations in some occasions. This is particularly true in presence of lazy evaluation, as is our case. We show with an example how the typically inefficient *generate-and-test* programming scheme of logic programming can be converted into a more efficient one if generation is done by means of a ND function, which is lazily evaluated according to the demand of the test. If the test is incremental and fails for the (partially generated) current candidate solution, then it is rejected without the need of completing its generation, and by backtracking a new candidate is tried. In a lazy functional language one could follow the same approach, but replacing backtracking by a (lazy) traversal of the (lazily generated) list of all solutions, which can be a very large intermediate structure.

A simple (but still with some interest) situation of this kind is the following: given a region  $R$  and a sequence of movements  $Mvs$ , we want to reorder  $Mvs$  into a new list of movements  $Mvs'$  whose sequential application to  $R$  avoids to meet any of the points of a list  $Pts$ . We first program as separate concerns the generation of candidate solutions (this is done by means of a ND function `permut Mvs`) and the check of validity of one candidate  $Mvs'$  (this is done by means of `check R Pts Mvs'`, whose definition uses some standard HO functions). If we had '*where-constructions*' in our language, we would then define the top level function as

```

solution R Pts Mvs = Mvs' <== check R Pts Mvs'
                    where Mvs' = permut Mvs % This is not true CFLP(R)

```

As it is well known, ‘*where-constructions*’ can be lifted, and this is done so in the  $CFLP(\mathcal{R})$ -program below, which uses the ND function // of Sect. 2.

```

permut [] = []
permut [X|Xs] = insert X (permut Xs) % It is ND, since insert is ND

insert X [] = [X]
insert X [Y|Ys] = [X,Y|Ys] // [Y|insert X Ys] % ND choice

check R Pts Mvs :- avoids Pts (doMoves R Mvs)
avoids Pts R = all (not.<<- R)) Pts % all and (.) are standard
%(doMoves R [Mv,Mv',Mv'',...]) is R `union` (Mv R) `union` (Mv' (Mv R))...
doMoves R [] = R
doMoves R [Mv|Mvs] = union R (doMoves Mvs (Mv R))

solution R Pts Mvs = solAux R Pts (permut Mvs)
%where
solAux R Pts Mvs = Mvs <== check R Pts Mvs

```

For instance, the goal

```

X == solution (rectangle (0,0) (2,2)) [(3,3)] [(homothety (0,0) 2),
      (homothety C 0.5),(translate (2,2)),(translate (1,1))],
      C <<- (rectangle (-1,-1) (0,0))

```

has three solutions, the first one being

```

C == (0, _A)
X == [(homothety (0, _A) 0.5), (translate (2, 2))
      (homothety (0, 0) 2), (translate (1, 1))]
{ _A >= -1, _A < -0}

```

It would be easy to generalize `solution` to a HO function implementing a generic *lazy generate-and-test* programming scheme, having as parameters the generator, the test and the parameters of the problem.

## 4 Execution mechanism

Like in CLP, constraints are independent of functions: when a constraint appears in a logic program the solver is invoked. In the same way, the constraint solver over reals is invoked when needed in  $CFLP(\mathcal{R})$ , while keeping the operational mechanism of the underlying functional logic language: lazy narrowing in our case. The computation strategy corresponds to the *demand driven strategy* (*dds*, for short) presented in [17], which is closely related to ‘needed narrowing’ [1], a strategy proved to be optimal for a distinguished class of term rewriting systems. *Dds* is based on the idea of evaluating subterms just when a demand exists. The demand is determined by the shape of the rules left hand sides (*lhs*, for short). The integration of constraints over real numbers does not require any modification on the above strategy. Constraint resolution comes into play when a rule is to be applied: the constraints must be satisfied. The ‘*compilation-to-Prolog*’ approach [9, 17] followed in the implementation of the language motivated us to take advantage of some existing  $CLP(\mathcal{R})$  system, instead of designing and implementing from scratch our own solver for real constraints. The main idea for managing real constraints is to manipulate them until we obtain

a suitable  $CLP(\mathcal{R})$  format, i.e. to evaluate the involved  $CFLP(\mathcal{R})$  expressions until we get  $CLP(\mathcal{R})$  expressions.

We concentrate here in FO computations, since we treat HO *à la Warren* [5, 23, 25], i.e., by translation to FO.

In the rest of this section, we start roughly presenting the *dds-strategy* for lazy narrowing, followed by some comments about the interaction with the  $CLP(\mathcal{R})$  constraint solver. We end observing that sharing cannot be regarded as an optional optimization, but as something needed in order to ensure the soundness of the answers.

#### 4.1 The Demand Driven Strategy for Lazy Narrowing

A formal presentation of this strategy is far out of the scope of this paper; the reader can find the details in [17]. All along this section, we use the rules  $R_{int} = \{R_i \equiv l_i = e_i \Leftarrow C_i \mid 1 \leq i \leq 4\}$  (numbered in textual order) for the function **intersct'** (see Sect. 3) to illustrate the strategy.

We start with some preliminary notions concerning the demanded positions in the *lhs* of the defining rules. Let  $R_f = \{R_i \mid 1 \leq i \leq m\}$  denote the set of the defining rules for a function symbol  $f$ . Let  $u$  denote a position. We say:

- $u$  is *demanded by the lhs of a rule*  $R_i$  iff it has a constructor at position  $u$ .
- $u$  is *demanded* iff  $u$  is demanded by the *lhs* of some rule  $R_i \in R_f$ .
- $u$  is *uniformly demanded* iff  $u$  is demanded by every *lhs* of the rules in  $R_f$ .

The study of the different kinds of demand in the defining rules of a function symbol  $f$  would lead to the definition of the *dds-strategy*. The defining rules are expressed by a *dds-tree* whose intended meaning is to reflect the strategy. In Fig. 1 we show the *dds-tree* related to the function **intersct'**, and in the following we try to explain the strategy by means of it.

Looking at the rules for **intersct'**, we observe the following situation:  $R_1$  and  $R_3$  demand position 1,  $R_2$  and  $R_3$  demand position 2, whereas  $R_4$  does not demand any position. A uniformly demanded position does not exist with respect to the whole set of rules. We proceed splitting  $R_{int}$  into three subsets:  $S_1 = \{R_1, R_3\}$ ,  $S_2 = \{R_2\}$ ,  $S_\emptyset = \{R_4\}$ , in such a way that position 1 is uniformly demanded by the rules in  $S_1$ , position 2 is uniformly demanded by the rules in  $S_2$ , and  $S_\emptyset$  contains a single rule that does not demand any position.

For each subset that uniformly demands a position  $i$ , the idea is that when the function is called, say **intersct'**  $e_1 e_2$ , the expression that occurs at position  $i$  must be evaluated to head normal form. Looking at the tree in Fig. 1, the first branch illustrates that position 1 is uniformly demanded. If evaluating  $e_1$  yields **emptyReg** or (**rectangle**  $_$   $_$ ) or a logic variable, the rules in  $S_1$  are applied. The first two results would lead to apply the appropriate rule ( $R_1$  or  $R_3$ ), while obtaining a variable as result would not allow to discard any of the rules: this suggests a choice point at the implementation level.

If after evaluating  $e_1$  we do not obtain any of the above results, or if another answer is asked for, we would continue with the second branch of the tree. Now, position 2 is uniformly demanded and the process is the same than above, except that in order to apply  $R_2$ , evaluating  $e_2$  has to yield **emptyReg** or a logic variable. Again, if it were not the case or if we asked for another answer, we would consider the third branch. This branch does not demand evaluation of the arguments and could be tried directly.

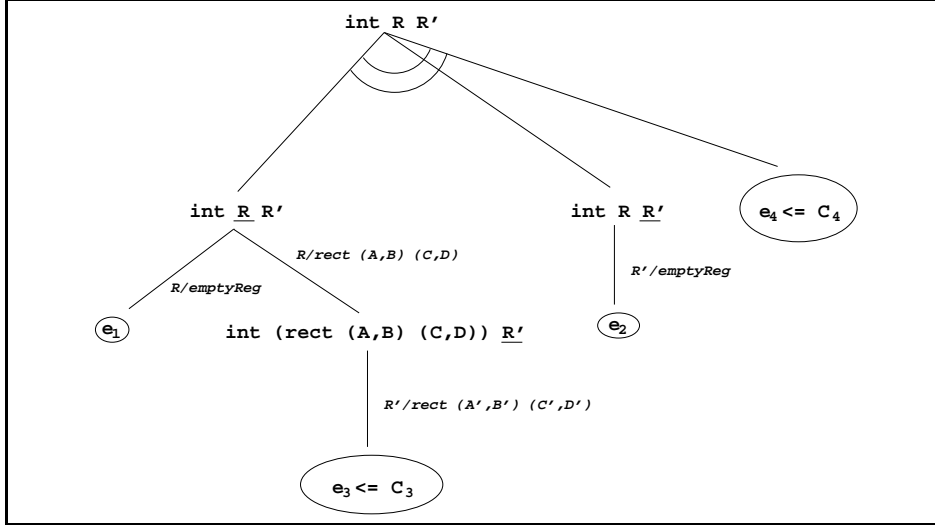


Figure 1: *dds-tree* for *intersct'*

At the implementation level, the strategy reflected by the *dds-tree* of Fig. 1 is expressed by means of the following *CLP*( $\mathcal{R}$ ) clauses:

```
int(R,R',H) :- hnf(R,HR), int1(HR,R',H).
int(R,R',H) :- hnf(R',HR'), int2(R,HR',H).
int(R,R',H) :- solve(C4), hnf(e4,H).
```

They represent the nodes at the first level of the tree.

```
int1(emptyReg,R',emptyReg).
int1((rectangle (A,B) (C,D)),R',H) :- hnf(R',HR'),
int1,2((rectangle (A,B) (C,D)),HR',H).
int2(R,emptyReg,emptyReg).
```

These clauses reflect the situation at the second level of nodes.

```
int1,2((rectangle (A,B) (C,D)),(rectangle (A',B') (C',D')),H) :-
solve(C3), hnf(e3,H).
```

Finally, this clause corresponds to the third level node.

There are two main 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*, which solves constraints (of rules and goals), and that is going to be explained in the next subsection.

The above code does not correspond exactly to the implementation, which is the result of many transformations and optimizations. Some of them are pointed out in Sect. 5.

## 4.2 Constraint solving

As it has been shown in the above example, the need for solving constraints appears in the rule application context. Prior to give the result of applying a given rule, we have to satisfy the constraints (if it were the case). This is done by means of the following predicate:

```

solve(( $\varphi$ ,  $\varphi'$ )) :- solve( $\varphi$ ), solve( $\varphi'$ ).
solve(L == R) :- hnf(L, L'), hnf(R, R'), equal(L', R').
solve(L / = R) :- hnf(L, L'), hnf(R, R'), notequal(L', R').
solve(L  $\diamond$  R) :- hnf(L, L'), hnf(R, R'), {L'  $\diamond$  R'}.
                    %  $\forall \diamond \in \{<, <=, >, >=\}$ 

```

The interaction with the  $CLP(\mathcal{R})$  solver is reflected in the last clause. Notwithstanding, every time a constraint (or an operation) over real expressions (i.e. with type real) appears the  $CLP(\mathcal{R})$  solver will be eventually invoked. The idea is always the same. The expressions have to be ‘simplified’ in order to let the  $CLP(\mathcal{R})$  system to solve the constraint. By simplifying we mean computing the head normal forms of both expressions. Doing this, we get a ‘simplified’ constraint written in the proper  $CLP(\mathcal{R})$  syntax, due to the fact that these *hnf* are going to be either logic variables or real numbers.

With strict equality (`==`) and disequality (`/=`) we follow the same ideas presented in [2, 19], apart from the straightforward extensions for dealing with real numbers.

## 4.3 Sharing

As pointed out in Sect. 3, nondeterministic functions are allowed in our framework. Owing to this fact, sharing is essential: it is not regarded as an optimization, but its incorporation turns to be inevitable as the following example (inspired in Hussmann [13]) shows.

```

coin = 0
coin = 1

double X = X + X

```

Consider the goal `(double coin)`. If sharing were not supported, we could obtain `1` as an answer, which is unsound for the ‘call-time choice’ semantics mentioned in Sect. 2. In the presence of sharing, the only two answers for the above goal are `0` and `2`. Both of them are sound.

## 5 Implementation issues.

In this section we show some aspects of the implementation of  $TOY(\mathcal{R})$ , specially those referring to constraints. The system is implemented in Sicstus Prolog 3.3, which provides a solver for real constraints developed by C. Holzbaur [12]. The translation of functions is basically the one presented in Sect. 4 with many optimizations, such as unfoldings, swapping arguments for a better indexing, elimination of unnecessary arguments, etc. Some of this optimizations (but not all) can be found in [17, 9].

$TOY(\mathcal{R})$  supports three kinds of constraints:

- Strict equalities,  $e == e'$ . They are treated as in [19, 17] with some new optimizations. The two most important are: a) when one of the expressions begins by a constructor symbol we imitate it, that is, we calculate a *head normal form* of the other, but oriented to the first one in order to fail as soon as possible if both terms are not equal. For instance, if we have  $e == (c \bar{e})$ , we first calculate  $hnf(e, c(\bar{X}))$ , where  $\bar{X}$  is a tuple of new variables. This optimization reduces the search space and has better termination properties in general. b) When both expressions are already in *head normal form*, we first look for a conflict of constructors, in order to fail as soon as possible without narrowing anything. At the same time, we build a 'continuation' (the set of constraints to be solved next), if no conflict is found. This implementation has better termination properties than a naive one, as shown by the example  $(c \textit{ loop } a) == (c \textit{ loop } b)$  (where  $c, a, b$  are constructor symbols and *loop* is an expression whose evaluation does not end). A naive strategy would decompose, proceeding sequentially with the new equalities  $\textit{loop} == \textit{loop}$  and  $a == b$ , thus incurring in non-termination, while our implementation is able to immediately fail because of the conflict  $a == b$ .
- Disequality,  $e \neq e'$ . Here no orientation is possible. Then both terms are narrowed to *head normal form* from the beginning (as in [19, 17]) and then the optimization b) for equality of *head normal forms* is suitable here (looking for conflict of constructors in order to get immediate success). Let us comment how  $\text{TOY}(\mathcal{R})$  deals with constraints of the form  $X \neq (c \bar{e})$ , like  $X \neq (\textit{suc } e)$  (where *zero* and *suc* are the constructors symbols for the type of natural numbers). If  $e$  is a pattern, then  $X \neq (\textit{suc } e)$  is in solved form and the disequality is stored. If  $e$  involves defined functions,  $\text{TOY}(\mathcal{R})$  does not perform, in a first alternative, any narrowing step over  $e$ , but guarantees the disequality binding  $X$  to *zero*. As a second alternative, to be eventually activated by backtraking,  $X \neq (\textit{suc } e)$  will be transformed into  $X == (\textit{suc } Y)$ ,  $Y \neq e$ , for which  $\text{TOY}(\mathcal{R})$  would need to narrow  $e$  in order to solve it. This is the natural way to implement disequalities with the philosophy of laziness and to preserve its semantics.
- Arithmetic constraints over reals, which, of course, include equality and disequality. The system has a group of predefined arithmetic functions which perform all the required numerical computations. The code for dealing with constraints over reals is isolated from the rest because all the calls to the solver appear into the code of these functions. For example, the function  $+$  has the following code:

$$+(X, Y, H) :- hnf(X, HX), hnf(Y, HY), \{H = HX + HY\} .$$

In presence of a disequality we must distinguish at run-time whether it is a syntactic one or it must be send to the solver.

The first and second kinds are in the base of the system and they were supported in a previous version.  $\text{TOY}(\mathcal{R})$  also implements the equality-function and disequality-function much more efficiently than the naive ones presented in Sect. 2 taking advantage of some optimizations.

We end this section with some comments about showing answers for goals in  $\text{TOY}(\mathcal{R})$ , which has been by far the more difficult task when incorporating constraints to  $\text{TOY}$  for obtaining  $\text{TOY}(\mathcal{R})$ . The main problems have come from the fact that Sicstus Prolog 3.3 does not provide good communication facilities with the solver (specially a predicate for

projecting constraints). We need the definition of *relevant variables* to describe how answers are presented. We say that a variable is relevant in an answer if: a) it appears in the goal, b) it appears in a term to which a variable of the goal is bound, or c) it appears in a non-linear constraint over real numbers. Using this concept the answer is formed by:

- Bindings (showed as equalities) for variables appearing in the goal.
- Relevant syntactic disequalities. A disequality is relevant if it contains some relevant variable in some of its terms.
- Constraints over real numbers that are calculated by making the projection over the set of relevant variables.

## 6 Conclusions

We hope to have convincingly demonstrated the interest and suitability of  $CFLP(\mathcal{R})$ , a functional logic programming language enhanced with the possibility of using real arithmetic constraints. Due to its functional component,  $CFLP(\mathcal{R})$  provides better tools, when compared to  $CLP(\mathcal{R})$ , for a productive declarative programming. Due to the use of constraints, the expressivity and capabilities of our language are clearly superior to those of a functional language. The language can be applied to a wide range of problems which include all  $CLP(\mathcal{R})$  applications and typical uses of functional programming for numerical algorithms.

For the execution mechanism of the language, we have easily integrated constraint solving into a sophisticated, state-of-the-art execution mechanism for lazy narrowing. This leads naturally to an implementation where  $CFLP(\mathcal{R})$ -programs are translated into a Prolog system equipped with a constraint solver. The merit of our approach, if any, is to show how easily existing constraint technology can be integrated into a functional logic framework. It seems clear to us that what have been done by us with linear real constraints can be realized also with other kind of interesting constraint systems, such as nonlinear constraints, constraints over finite domains, or boolean constraints.

**Acknowledgements:** We thank Christian Holzbaur for kindly helping us with some aspects of his constraint solver; we are indebted to Puri Arenas, Rafa Caballero and Juan Carlos González for their help while developing and writing the work. This research has been partially supported by the Spanish National Project TIC95-0433-C03-09 “CPD” and by the Esprit BRA Working Group EP-22457 “CCL II”.

## References

- [1] Antoy S., Echahed R., Hanus M.: *A Needed Narrowing Strategy*. 21st ACM Symp. on Principles of Programming Languages, 268–279, Portland 1994.
- [2] Arenas-Sánchez P., Gil-Luezas A., López-Fraguas F.J.: *Combining Lazy Narrowing with Disequality Constraints*. Procs. of PLILP’94, Springer LNCS 844, 385–399, 1994.
- [3] Arenas-Sánchez P., Rodríguez-Artalejo M.: *A Semantic Framework for Functional Logic Programming with Algebraic Polymorphic Types*. Procs. of CAAP’97, to appear, 1997.
- [4] Darlington J., Guo Y.K.: *A New Perspective on Integrating Functions and Logic Languages*. Procs. of the 3rd Conference on Fifth Generation Computer Systems, Tokyo, 682–693, 1992.

- [5] González-Moreno J.C.: *A Correctness Proof for Warren's HO into FO Translation*. Procs. of GULP'93, 569–585, 1993.
- [6] González-Moreno J.C., Hortalá-González T., Rodríguez-Artalejo M.: *On the Completeness of Narrowing as the Operational Semantics of Functional Logic Programming*. Procs. of CSL'92, Springer LNCS 702, 216–230, 1993.
- [7] González-Moreno J.C., Hortalá-González T., López-Fraguas F.J, Rodríguez-Artalejo M.: *A Rewriting Logic for Declarative Programming*. Procs. of ESOP'96, Springer LNCS 1058, 156–172, 1996.
- [8] González-Moreno J.C., Hortalá-González T., Rodríguez-Artalejo M.: *A Higher Order Rewriting Logic for Functional Logic Programming*. Procs. of ICLP'97, to appear, 1997.
- [9] Hanus M.: *Efficient Translation of Lazy Functional Logic Programs into Prolog*. Procs. of LOPSTR'95, Springer LNCS 1048, 252–266.
- [10] Hanus M.: *The Integration of Functions into Logic Programming: A Survey*. Journal of Logic Programming 19-20. Special issue “Ten Years of Logic Programming”, 583–628, 1994.
- [11] *Report on the Programming Language Haskell: a Non-strict, Purely Functional Language*. Version 1.4, Peterson J. and Hammond K. (eds.), January 1997.
- [12] Holzbaaur C.: *OFAI clp(Q,R) Manual*. Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.
- [13] Hussmann H.: *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser, 1993.
- [14] Jaffar J., Lassez J.L.: *Constraint Logic Programming*. Procs. of the 14th ACM Symp. on Principles of Programming Languages, 114–119, Munich 1987.
- [15] Jaffar J., Maher M.J.: *Constraint Logic Programming: A Survey*. Journal of Logic Programming 19/20, 503–582, 1994.
- [16] Jaffar J., Michaylov S., Stuckey P.J., Yap R.H.C.: *The CLP(R) Language and System*. ACM Transactions on Programming Languages and Systems, Vol. 14, No. 3, 339–395, July 1992.
- [17] Loogen R., López-Fraguas F.J, Rodríguez-Artalejo M.: *A Demand Driven Computation Strategy for Lazy Narrowing*. Procs. of PLILP'93, Springer LNCS 714, 184–200, 1993.
- [18] López-Fraguas F.J.: *A General Scheme for Constraint Functional Logic Programming*. Procs. of ALP'92, Springer LNCS 632, 213–217, 1992.
- [19] López-Fraguas F.J.: *Programación funcional y lógica con restricciones*. PhD thesis, DIA-UCM, Madrid 1994.
- [20] Mandel L.: *Constrained Lambda Calculus*. Shaker, 1995.
- [21] Moreno-Navarro J.J., Rodríguez-Artalejo M.: *Logic Programming with Functions and Predicates: The Language BABEL*. Journal of Logic Programming 12, 189–223, 1992.
- [22] Mück A., Streicher T., Lock H.: *A Tiny Constraint Functional Logic Language and Its Continuation Semantics*. Procs. of ESOP'94, Springer LNCS 788, 439–453, 1994.
- [23] Naish L.: *Higher-order Logic Programming in Prolog*. Procs. of the JICSLP'96 Post-Conference Workshop “Multi-Paradigm Logic Programming”, Chakravarty M.M.T., Guo Y. and Ida T. (eds), Report 96–28, Technische Universität Berlin, September 1996.
- [24] Pfenning F.: *Types in Logic Programming*. The MIT Press, 1992.
- [25] Warren D.H.D.: *Higher-order extensions to Prolog: are they needed?*. Hayes J.E., Michie D. and Pao Y-H. (eds), *Machine Intelligence 10*, Ellis Horwood, 441–454, 1982.