Real Constraints within a Functional Logic Language*

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

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

Abstract

We present a declarative language – $CFLP(\mathcal{R})$ – which integrates lazy functional programming, logic programming and constraint solving over real numbers. Both a (higher order, polymorphic, lazy) functional language and (pure) $CLP(\mathcal{R})$ can be isolated as subsets of our language. Through several examples we attempt to demonstrate the interest of $CFLP(\mathcal{R})$. The execution mechanism of the language consists of a combination of lazy narrowing and constraint solving. For the purpose of giving a specification of such operational semantics, but serving also as a very simple method for implementing the language, we propose a translation of $CFLP(\mathcal{R})$-programs into a logic programming language supporting real arithmetic 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 languages (see [12] for a survey). Most of the interest in this field started with the proposal of the $CLP(\mathcal{X})$ scheme [11], 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 $CLP(\mathcal{R})$ [13], which extended traditional logic programming 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.

Another important branch in the evolution of declarative languages has been the integration of the functional and logic programming (FLP) paradigms (see [8] for

*This research has been partially supported by the "U.C.M precompetitivo 95/5525" and the Spanish National Project TIC95-0433-C03-09 "CPD".

a survey). To link together these two independent branches of evolution of logic programming appears as a natural interesting task. In [16, 17] a general scheme – $CFLP(\mathcal{X})$ – for *constraint functional logic programming* (CFLP) was proposed, with the aim of extending lazy functional logic programming (in the sense of, e.g., [18]) in the same way that $CLP(\mathcal{X})$ extended traditional logic programming. There are other proposals for CFLP (see e.g. [17] for commented references) but, as far as we know, they have not fructified in the development of concrete, practical languages, maybe with the exception of [1] where a lazy FLP language is extended to cope with disequality constraints over syntactic terms (see [17] for a thorough exposition of that language and the $CFLP(\mathcal{X})$ scheme). We have recently known of a work [4], still in quite an early stage of development, where the functional logic language BABEL [18] is extended with real arithmetic constraints, resulting in a language similar to ours.

In this paper we investigate the language $CFLP(\mathcal{R})$ which incorporates real arithmetic constraints to a higher order lazy functional logic language in the spirit of [6]. Two *a priori* reasons encouraged us to start our work: the success of $CLP(\mathcal{R})$ and well-known classical examples [10] showing the expressiveness of lazy functional languages for programming numerical algorithms.

The rest of the paper is organized as follows: in Sect. 2 we describe the language and its syntax. Section 3 contains examples showing some of the possibilities of our language. In Sect. 4 we explain the execution mechanism of the language, by means of a translation of $CFLP(\mathcal{R})$-programs into a constraint logic programming language. Finally, Sect. 5 summarizes some conclusions.

2 Description of the language

Our language can be thought as an extension of a higher order lazy functional logic language in the spirit of [6], equipped with a polymorphic type system and, most importantly, with the capability of using real arithmetic constraints in programs and constraint solving as a part of the execution mechanism.

We detail now the syntax. We consider the following sets of symbols and syntactic constructions:

• $tc, tc_1, \ldots \in TCS$: *type constructor* symbols, each with associated arity. We will use TCS^n for the set of type constructor symbols of arity n (and similarly for other sets of symbols below). We assume that TCS includes $real/0$, and $[\cdot]/1$ (list type constructor).

• TCS, together with a set of type variables $\alpha, \beta, \ldots \in TVar$, determine the set of *polymorphic types* $\tau, \tau_1, \ldots \in Type$ defined by

$$\tau ::= \alpha \mid tc/0 \mid (tc/n\ \tau_1 \ldots \tau_n) \mid (\tau_1, \ldots, \tau_n) \mid (\tau_1 \to \tau_2)$$

We assume, as usual, that $\to$ is right associative. The first four alternatives in the previous definition determine the set $CType$ of *constructed types*.

• $c, c_1, \ldots \in CS$: *data constructor* symbols, each with associated type $\tau_1 \to \ldots \to \tau_n \to \tau$ ($\tau, \tau_i \in CType$). n is the arity of the constructor symbol. We assume that CS includes $[\,]/0, [\cdot|\cdot]/2$ (empty list and list data constructor). We assume, as usual, that

type and data constructor symbols are introduced by means of datatype declarations in a Haskell-like notation.

• $RFS = \{+/2, -/2, -/1, */2, //2\}$ is the set of *primitive real function* symbols, which have type $real \to real \to real$ (except $-/1$, which has type $real \to real$). In addition, we consider a set $Real$ of suitable representations (floating point, for instance) of (a set of) real numbers. Elements $a, b, \ldots$ of $Real$ have type $real$.

• $f, g, \ldots \in FS$: *user defined function* symbols, each with associated *program arity* n and type $\tau_1 \to \ldots \to \tau_m \to \tau$ ($\tau, \tau_i \in Type$, τ does not contain $\to$). m is the *type arity* of the function symbol, and must be $m \geq n$. The program arity of f expresses the number of arguments that f requires in order to be evaluated. By $f \in FS^n$ we mean that the program arity of f is n.

• Given a set $X, Y, \ldots \in Var$ of data variables, the set of *expressions* $e, l, r, \ldots \in Exp$ is defined by

$$e ::= X \mid a \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e_1\ e_2)$$

where $a \in Real, c \in CS, f \in FS \cup RFS$. We assume, as usual, that application associates to the left, and we omit parentheses accordingly. We require expressions to be well-typed, but we skip the description of well known type checking or type inference techniques. An expression is *primitive* if no user defined function appears in it. An important class of expressions is that of (first order) *patterns* $t, s \in Pat$, defined by

$$t ::= X \mid a \mid c\ t_1 \ldots t_n \mid (t_1, \ldots, t_n)$$

where $c \in CS^n$, $a \in Real$.

• $\Pi = \{== /2\} \cup RRS$ is the set of *primitive relation* symbols, where RRS, the set of real such symbols, is $\{< /2, > /2, =< /2, >= /2, =\backslash= /2\}$. Equality $==$ must be interpreted as *strict equality* (see e.g [18]).

• An *atomic constraint* $at \in Atom$ takes the form $e_1 \diamondsuit e_2$ where $\diamondsuit \in \Pi$. e_1, e_2 must be of type $real$ if $\diamondsuit \in RRS$, and simply of the same (first order) type if $\diamondsuit$ is $==$. A *constraint* $\varphi, \psi, \ldots \in Con$ is a multiset of atomic constraints, written in the form $at_1, \ldots, at_n$, and should be interpreted as a conjunction. A constraint is *primitive* if so are all the involved expressions.

• A $CFLP(\mathcal{R})$-*program* consists of *rules* for defining the symbols $f \in FS$. Each $f \in FS^n$ has one or more rules of the form

$$f\ t_1 \ldots\ t_n = e \Leftarrow \varphi$$

where $t_i \in Pat, e \in Exp, \varphi \in Con$. $t_1 \ldots t_n$ must be linear, i.e., no variable can occur more than once. Each rule has a conditional reading: the value of $f\ t_1 \ldots\ t_n$ (the head) is the value of e (the body) if φ (the constraint) is satisfied. Variables occurring only in φ have existential reading. The condition $\Leftarrow \varphi$ may be omitted if φ is empty.

Rules must also satisfy some nonambiguity conditions, ensuring the functional nature of definitions. See [17] for an abstract semantic notion of nonambiguity which can be strengthened to decidable conditions similar to that in [18]. Types for functions can be inferred from their definitions or can be declared in a Haskell-like notation.

Observe that $CFLP(\mathcal{R})$ has (the core of) a lazy functional language as a subset. Our language does not contemplate explicitly the possibility of defining *predicates* through clauses, but this is not a lack of expressiveness since $true$-valued functions can be used instead. For instance, in $CLP(\mathcal{R})$ clauses take the form $p(t_1, \ldots, t_n) :- b_1, \ldots, b_m, \varphi$ and can be straightforwardly[1] translated into our language as rules

$$p\ t_1 \ldots\ t_n = true \Leftarrow b_1 == true, \ldots, b_m == true, \varphi$$

With this translation in mind, any $CLP(\mathcal{R})$-program can be seen as a $CFLP(\mathcal{R})$-program.

• A *goal* is simply a constraint φ, whose variables have an existential reading. Computed answers $\langle \theta, \varphi \rangle$ consist of a substitution θ and a primitive real constraint φ in some kind of solved form. The operational mechanism for solving goals is specified in Sect. 4, and basically is a combination of lazy narrowing and constraint solving. Laziness allows to define and compute with infinite objects. With respect to real arithmetic constraint solving, we stick to $CLP(\mathcal{R})$ philosophy: linear constraints are checked for satisfiability and simplified, while non-linear constraints are frozen until they become instantiated enough.

Our approach to HO functions and computations is borrowed from [5, 6], relying in a first order view of HO features. In this treatment, all expressions are translated into first order expressions, by introducing new constructor symbols for the case of partial applications (of functions and constructors) and a new operation @ (equipped with suitable rules) for applications which cannot be expressed by means of the (enhanced) set of constructor and function symbols (see [5] for details). The approach allows the use of HO logic variables in programs and goals. The operational semantics described in Sect. 4 assumes this FO translation of programs.

To give a semantic characterization of $CFLP(\mathcal{R})$ is far out of the scope of this paper. Some words in order to clarify the picture: in [17] it is explained how the untyped first order subset of $CFLP(\mathcal{R})$ can be characterized as an instance of the $CFLP(\mathcal{X})$ scheme, from which it inherits a least model declarative semantics, as well as soundness and completeness results for the operational semantics. [17] also shows that the FO approach to HO functional logic programming of [5] fits also the $CFLP(\mathcal{X})$ scheme. Semantic issues related to polymorphic types are subject of work in progress, but the combination of all the pieces of the puzzle still remains to be done.

3 Some programming examples

As pointed out in Sect. 2, any $CLP(\mathcal{R})$-program can be seen as a $CFLP(\mathcal{R})$-program, thus determining a wide range of applications for our language. We will not insist here on this matter, but prefer instead to concentrate on the extra capabilities of the language. Our first example refers to complex arithmetic and shows the high degree of flexibility that $CFLP(\mathcal{R})$ provides to functional programs. A second part

[1] If $t_1 \ldots t_n$ is not linear, an additional easy transformation is needed.

of the example shows the benefits of mixing functional and logic styles, to which we can add the power of constraints for search pruning. Our last example demonstrates the usefulness of combining lazy functions with constraint solving over real numbers, exploiting lazy evaluation over infinite data structures.

3.1 Complex numbers

We represent a complex number $a + bi$, $a, b \in Real$ as a tuple (a, b). We can express this in $CFLP(\mathcal{R})$ by means of the type alias declaration:

$type\ complex = (real, real)$

The definitions of functions for the arithmetic operations over complex numbers are the natural ones. For example, complex multiplication would be:

$c\_times :: complex \to complex \to complex$
$c\_times\ (R1, I1)\ (R2, I2)\ = (R1 * R2 - I1 * I2, R1 * I2 + R2 * I1)$

A goal which makes a "functional" use of c_times is $c\_times\ (5, 6)\ (2, 4)\ == H$ for which we obtain as computed answer $H = (-14, 32)$. But notice that our language also allows to make a "logical" use of c_times, as happens in the goal $c\_times\ (5, 6)\ Y\ == (2, 4)$ for which we obtain the answer $Y = (0.55737, 0.13114)$.

Answers can include some constraint in solved form, as in the case of the goal $c\_times\ (X, Y)\ (0, 2)\ == (Z + 1, 4)$ for which the answer consists of the binding $X = 2$ together with the linear constraint $Z + 2Y + 1 = 0$. As in $CLP(\mathcal{R})$, the goal $c\_times\ (X, 1)\ (1, Y)\ == (4, 4)$ can not be fully solved because the resolution of the goal generates not only the linear constraint $Y = -4 + X$, but also the non-linear equation $3 - X * Y = 0$.

The use of "predicates" and non-determinism allows to formulate some problems in a clean way, as in the following case: imagine we want to recognize if n complex numbers (considered as points in the plane) are the vertices of a regular polygon. This is equivalent to the fact that, for an appropriate reordering $z_1, \ldots, z_n$ of the points, the sequence $z_2 - z_1, z_3 - z_2, \ldots, z_1 - z_n$ (the oriented edges of the polygon) forms a geometric progression with ratio $R = a + bi$, such that $|R| = 1$, i.e. $a^2 + b^2 = 1$. We can define the function:

$$
\begin{array}{lll}
regular\_polygon :: [complex] \to bool & & \\
regular\_polygon\ Zs\ = true & \Leftarrow & permutation\ Zs\ Zs' == true, \\
 & & geometric\ (edges\ Zs')\ (A, B) == true, \\
 & & A * A + B * B == 1
\end{array}
$$

where $permutation$ is defined in the standard Prolog way:

$$
\begin{array}{lll}
permutation :: [A] \to [A] \to bool & & \\
permutation\ [\,]\ [\,]\ = true & & \\
permutation\ Xs\ [X|Xs'] = true & \Leftarrow & select\ Xs\ X\ Ys == true, \\
 & & permutation\ Ys\ Xs' == true \\
 & & \\
select :: [A] \to A \to [A] \to bool & & \\
select\ [X|Xs]\ X'\ Xs'\ = true & \Leftarrow & X' == X, Xs' == Xs \\
select\ [X|Xs]\ Y\ [X'|Zs]\ = true & \Leftarrow & X' == X, select\ Xs\ Y\ Zs == true
\end{array}
$$

The function *geometric* recognizes if a list Zs is a geometric progression with ratio R, and it can be defined by:

$$
\begin{array}{l}
geometric :: [complex] \rightarrow complex \rightarrow bool \\
geometric\ [Zs]\ R\ = true \\
geometric\ [Z1, Z2|Zs]\ R\ = true \quad \Leftarrow \quad Z2 == c\_times\ Z1\ R, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ geometric\ [Z2|Zs]\ R == true
\end{array}
$$

A way of defining *edges* that many functional programmers would like is the following:

$$
\begin{array}{l}
edges :: [complex] \rightarrow [complex] \\
edges\ [Z|Zs]\ = zipWith\ c\_minus\ (append\ Zs\ [Z])\ [Z|Zs]
\end{array}
$$

where *c_minus* and *append* are defined in the natural way. *zipWith* is the (slightly modified) standard HO function:

$$
\begin{array}{l}
zipWith :: (A \rightarrow B \rightarrow C) \rightarrow [A] \rightarrow [B] \rightarrow [C] \\
zipWith\ F\ [\,]\ [\,]\ =\ [\,] \\
zipWith\ F\ [X|Xs]\ [Y|Ys]\ =\ [F\ X\ Y|zipWith\ F\ Xs\ Ys]
\end{array}
$$

A possible goal for this program would be $regular\_polygon\ [(0,0),(1,1),(0,1),P] == true$, for which we obtain $P = (1,0)$ as answer.

Of course it can be argued that *regular_polygon*, although declaratively clean, computes very inefficiently due to its *generate-and-test* structure. With little effort we can do it better, by testing before generating —a common point of view in CLP—. The improved function could be:

$$
\begin{array}{lll}
regular\_polygon\ Zs\ = true & \Leftarrow & same\_length\ Zs\ Zs' == true, \\
 & & geometric\ (edges\ Zs')\ (A,B) == true, \\
 & & A * A + B * B == 1, \\
 & & permutation\ Zs\ Zs' == true
\end{array}
$$

where *same_length* is defined as:

$$
\begin{array}{l}
same\_length :: [A] \rightarrow [B] \rightarrow bool \\
same\_length\ [\,]\ [\,]\ = true \\
same\_length\ [X|Xs]\ [Y|Ys] = true \quad \Leftarrow same\_length\ Xs\ Ys == true
\end{array}
$$

3.2 Solving equations by the iteration method

We will program the *iteration method*, one of the simplest methods for solving numerical equations over a single variable. In this method, for solving an equation $x = f(x)$, a sequence of approximations $x_0, x_1, x_2, \ldots$ is generated, starting from an arbitrarily chosen initial x_0. The sequence is given by $x_0, x_1 = f(x_0), x_2 = f(x_1), x_3 = f(x_2), \ldots$ Under certain conditions over f and x_0, which we will not discuss here, it is ensured that this sequence converges to a solution of the equation $x = f(x)$. As an estimation of the (signed) error of the approximation x_i we can use $\epsilon_i \equiv x_i - x_{i-1}$.

For obtaining the infinite list of approximations $[x_0, x_1, x_2, \ldots]$ we can use the following polymorphic function, which is standard in lazy functional programming.

$$
\begin{array}{l}
iterate :: (A \rightarrow A) \rightarrow A \rightarrow [A] \\
iterate\ F\ X = [X|iterate\ F\ (F\ X)]
\end{array}
$$

The following function *delta* transforms a (possibly infinite) list of real numbers $[x_0, x_1, x_2, \ldots]$ into the list of differences $[x_1 - x_0, x_2 - x_1, \ldots]$.

$delta :: [real] \rightarrow [real]$
$delta\ [X0, X1|Xs] = [X1 - X0|delta\ [X1|Xs]]$
$delta\ [X0] = [\,]$

The next function *accurate_sequence* attaches to every approximation its error estimation, i.e., $accurate\_sequence\ [x_0, x_1, x_2, \ldots] = [(x_1, x_1 - x_0), (x_2, x_2 - x_1), \ldots]$. Another standard list processing function (*zip*) is used in the definition:

$accurate\_sequence :: [real] \rightarrow [(real, real)]$
$accurate\_sequence\ [X|Xs] = zip\ Xs\ (delta\ [X|Xs])$

$zip :: [A] \rightarrow [B] \rightarrow [(A, B)]$
$zip\ [\,]\ [\,] = [\,]$
$zip\ [X|Xs]\ [Y|Ys] = [(X, Y)|zip\ Xs\ Ys]$

Now we can define the function *accurated_iterations* which, for given f and x_0, returns the (infinite) list $[(x_1, \epsilon_1), (x_2, \epsilon_2), \ldots]$ of successive approximations paired with their corresponding error estimation.

$accurated\_iterations :: (real \rightarrow real) \rightarrow real \rightarrow [(real, real)]$
$accurated\_iterations\ F\ X0 = accurate\_sequence\ (iterate\ F\ X0)$

For accessing to particular pairs (x_i, ϵ_i) we can use the standard function

$nth\ 1\ [X|Xs] = X$
$nth\ N\ [X|Xs] = nth\ (N - 1)\ Xs \Leftarrow N > 1$

The program needs, of course, one or more functions to which apply the iteration method. We assume, for instance, that two of them are defined in the program:

$f\ X = 1/(2 + X * X)$
$g\ X = (2 + X)/(10 + X * X * X)$

A possible goal corresponding to a "functional" use of the program could be

$nth\ 3\ (accurated\_iterations\ f\ 0) == Approx$

where we ask for the result (a value of *Approx* in the form of a pair (x_3, ϵ_3)) corresponding to three steps of the iteration method applied to the equation $x = 1/(2 + x^2)$, starting from an initial value $x_0 = 0$. As an answer, we would obtain $Approx = (0.455056, 0.010611)$.

Again, as in the first example, our language allows a more flexible use of the program. For example, we can also pose the goal $nth\ N\ (accurated\_iterations\ f\ 0) == Approx$. In this case we would obtain, in successive answers generated by backtracking, the approximations corresponding to increasing number of steps

$N = 1, Approx = (0.5, 0.5)$ $\quad$ $N = 2, Approx = (0.444444, -0.055555)$
$N = 3, Approx = (0.455056, 0.010611)$ $\quad$ $N = 4, Approx = (0.453088, -0.001968)$ $\quad \ldots$

More interesting is the goal

$nth\ N\ (accurated\_iterations\ f\ 0) == (XN, Eps), Eps < 0.01, -Eps < 0.01$

similar to the previous one, but with an additional restriction over the error. In this case, we expect to obtain a number of steps N great enough for producing an approximation (x_n, ϵ) satisfying the error bound $|\epsilon| < 0.01$ (expressed through the inequalities $\epsilon < 0.01, -\epsilon < 0.01$). An answer to this goal would be $N = 4, X = 0.453088, Eps = -0.001968$. Notice that the order of the constraints in the goal is not

important. It could be equally solved if the conditions $Eps < 0.01, -Eps < 0.01$ were prior to the remaining equality.

In this last example HO functions (like *iterate*) are mostly intended to be used in a functional manner, in the sense that their HO arguments are fully instantiated when calling to the HO function. But this is not mandatory. Due to our treatment of HO functions, nothing avoids to consider goals with HO variables. The answers for these goals would include bindings for those HO variables. For instance, we could propose the goal $nth\ 2\ (accurated\_iterations\ F\ 0)\ ==\ (X, Eps),\ Eps < 0.05,\ -Eps < 0.05$, asking for a function F such that two steps of the iteration method starting in 0 produce an approximation X with absolute error estimation less than 0.05. An answer to this goal would be $F = g, X = 0.219824, Eps = 0.019824$.

4 Specification of the operational semantics

The specification for lazy narrowing with constraints over reals is conceived as a compilation to the CLP system over rationals or reals clp(Q,R) [9] running on a Prolog system extended with attributed variables. We have chosen ECL^{i}PS^e 3.5.2 [3] as the Prolog extension based on attributed variables[2] on which clp(Q,R) is running. In our examples we only use clp(Q,R) over reals, but rationals could be used instead. In the rest of the paper, we call this combination ECLP(R). The clp(Q,R) constraint solving mechanism is invoked by means of enclosing the constraints by the delimiters '{', '}' (in that way constraints are added to the current store and checked for satisfiability).

By the use of the clp(Q,R) solver we have alleviated ourselves of the task of designing and implementing our own solver for real constraints. This is quite in the spirit of the "compilation-to-Prolog" approach in which, for instance, unification and substitutions are not implemented from the scratch, but Prolog's mechanisms are profited instead. Observe also that to replace clp(Q,R) by a different solver would cause no problem to our implementation (which can be thought as *parameterized* by the solver).

The specification translates any given $CFLP(\mathcal{R})$-program P into a set of clauses $ETC(P)$. The ETC translation may be chosen to perform different strategies for lazy constrained narrowing. Our approach is a natural extension of the "compilation-to-Prolog" way of implementing lazy narrowing [2, 7, 14, 15]. More concretely, our work is based on [15], where the reader can find more details about the translation into Prolog of different strategies. When introducing the specification we use the simplest one for the sake of clarity: the *naive* strategy.

Within $ETC(P)$ we represent $CFLP(\mathcal{R})$ variables and expressions as Prolog variables and terms. Constraints can also be represented as Prolog terms (using '==', ',' and the RRS symbols as infix operators). The computed answers $\langle \theta, \varphi \rangle$ are not made explicit by our implementation; they are subsumed by ECL^{i}PS^e-unification and the clp(Q,R) constraint solving mechanism. $ETC(P)$ consists of clauses for two main

[2] Attributed variables are a special ECL^{i}PS^e data type which represents variables with some attached attributes. It allows the system's behaviour on unification to be customized. In most situations, an attributed variable behaves like a normal variable. E.g. it can be unified with other terms and **var/1** succeeds on it. For more details, see [3].

predicates: *hnf*, which narrows expressions into *head normal form*, and *solve*, which solves constraints (of rules and goals).

4.1 Computation of Head Normal Forms

The predicate $hnf(E, H)$ specifies that H is one of the possible results of narrowing the expression E into head normal form. Predicates $\#f$ correspond to the defined function symbols and they are defined in the next subsection.

Clauses for *hnf*

$hnf(E, H)$	:-	$var(E), !, H = E.$	
$hnf(E, H)$	:-	$number(E), !, H = E.$	
$hnf((E_1, \ldots, E_m), H)$	:-	$!, H = (E_1, \ldots, E_m).$	
$hnf(c(E_1, \ldots, E_m), H)$	:-	$!, H = c(E_1, \ldots, E_m).$	$\% \ \forall \ c \in CS^m$
$hnf(f(E_1, \ldots, E_n), H)$	:-	$!, \#f(E_1, \ldots, E_n, H).$	$\% \ \forall \ f \in FS^n$
$hnf(\diamond E, H)$	:-	$!, hnf(E, E'), \{H = \diamond E'\}.$	$\% \ \forall \ \diamond \in RFS^1$
$hnf(E_1 \diamond E_2, H)$	:-	$!, hnf(E_1, E_1'), hnf(E_2, E_2'), \{H = E_1' \diamond E_2'\}.$	$\% \ \forall \ \diamond \in RFS^2$

Note that in the case of expressions of type *real*, computing *hnf* is the same as computing normal form, since reduction to a primitive expression (like 3 or $7 + X$) is performed.

4.2 Rule Application

For each (FO translation of a) defining rule of a $CFLP(\mathcal{R})$-program P

$$f(t_1, \ldots, t_n) \ = \ e \Leftarrow \varphi$$

we define an associated ECLP(R)-clause

$$\#f(E_1, \ldots, E_n, H) \quad :- \quad unify(E_1, t_1), \ldots, unify(E_n, t_n), solve(\varphi), hnf(e, H).$$

We use $unify(E, T)$ to unify the expression E and the linear term T, reducing E by lazy narrowing as much as demanded by the constructors occurring in T.

Clauses for *unify*

$unify(E, X)$	:-	$var(X), !, X = E.$	
$unify(E, X)$	:-	$number(X), !, hnf(E, E'), \{E' = X\}.$	
$unify(E, (T_1, \ldots, T_m))$	:-	$!, hnf(E, (E_1, \ldots, E_m)), unify(E_1, T_1), \ldots, unify(E_m, T_m).$	
$unify(E, c(T_1, \ldots, T_m))$	:-	$!, hnf(E, c(E_1, \ldots, E_m)), unify(E_1, T_1), \ldots, unify(E_m, T_m).$	
		$\% \ \forall \ c \in CS^m, m \geq 0$	

4.3 Goal Solving with Incremental Occur Check

The constraint solver *solve* handles the constraints of rules and goals. In clause $solve(L == R)$, which depends on the predicate $==\_hnf(L, R)$, the strict equation between expressions L and R is solved by lazy narrowing, following [15]. For the rest of the clauses $solve(L \diamondsuit R)$, the constraint $L \diamondsuit R$ is managed by first evaluating L and R to *head normal form* L' and R' respectively and then letting the clp(Q,R) system to solve the constraint $\{L' \diamondsuit R'\}$.

Clauses for *solve*

$solve((\varphi, \varphi')) \quad :- \quad !, solve(\varphi), solve(\varphi').$
$solve(L \Diamond R) \quad :- \quad !, hnf(L, L'), hnf(R, R'), \Diamond\_hnf(L', R'). \qquad \% \ \forall \ \Diamond \in \{==\} \cup RRS$

Clauses for $\Diamond$*_hnf*

$\Diamond\_hnf(L, R) \quad :- \quad \{L \Diamond R\}. \qquad \% \ \forall \ \Diamond \in RRS$
$==\_hnf(X, Y) \quad :- \quad (is\_real\_hnf(X); is\_real\_hnf(Y)), !, \{X = Y\}.$
$==\_hnf(X, H) \quad :- \quad var(X), !, bind(X, H).$
$==\_hnf(H, X) \quad :- \quad var(X), !, bind(X, H).$
$==\_hnf((L_1, \ldots, L_m), (R_1, \ldots, R_m)) \quad :- \quad !, solve(L_1 == R_1), \ldots, solve(L_m == R_m).$
$==\_hnf(c(L_1, \ldots, L_m), c(R_1, \ldots, R_m)) \quad :- \quad !, solve(L_1 == R_1), \ldots, solve(L_m == R_m).$
$\% \ \forall \ c \in CS^m, m \geq 0$

Clauses for *is_real_hnf*

The first clause of $==\_hnf$ uses the predicate *is_real_hnf*. Its intended use is to distinguish *head normal forms* standing for admissible clp(Q,R) expressions, for which there are only two possibilities: a number or an attributed variable (e.g. the variable H with attribute $3 + X$). The latter case is checked in ECL^{i}PS^e by means of the type-checking predicate **meta/1**.

$is\_real\_hnf(X) \quad :- \quad number(X), !.$
$is\_real\_hnf(X) \quad :- \quad meta(X).$

Clauses for *bind* **and** *occurs_not*

The second and third clauses of $==\_hnf$ use *bind(X, H)*. This predicate forces the expression H to be completely evaluated (to normal form) and binds the variable X to the result, taking into account the occur check. The evaluation of H, the binding of X and the occur check are interleaved according to the following specification.

$bind(X, H) \quad :- \quad is\_real\_hnf(H), !, \{X = H\}.$
$bind(X, H) \quad :- \quad var(H), !, X = H.$
$bind(X, (E_1, \ldots, E_m)) \quad :- \quad !, occurs\_not(X, E_1), \ldots, occurs\_not(X, E_m),$
$X = (X_1, \ldots, X_m), hnf(E_1, H_1), bind(X_1, H_1), \ldots,$
$hnf(E_m, H_m), bind(X_m, H_m).$
$bind(X, c(E_1, \ldots, E_m)) \quad :- \quad !, occurs\_not(X, E_1), \ldots, occurs\_not(X, E_m),$
$X = c(X_1, \ldots, X_m), hnf(E_1, H_1), bind(X_1, H_1), \ldots,$
$hnf(E_m, H_m), bind(X_m, H_m). \qquad \% \ \forall \ c \in CS^m, m \geq 0$

$occurs\_not(X, Y) \quad :- \quad var(Y), !, X \backslash== Y. \qquad \% \ X \backslash== Y$ checks syntactic disequality
$occurs\_not(X, c(E_1, \ldots, E_m)) \quad :- \quad !, occurs\_not(X, E_1), \ldots, occurs\_not(X, E_m).$
$\% \ \forall \ c \in CS^m, m \geq 0.$
$occurs\_not(X, E).$

4.4 Optimizations

First, we observe that calls to some predicates can be easily **unfolded by partial evaluation**. These predicates are *unify*, *solve* and some particular uses of $hnf(E, H)$, for instance when E is already in head normal form, or when E is a Prolog term representing a function call. After performing this optimization the rule application clauses for a part of the example given in Sect. 3.2 are:

```
nth(N,Ys,H) :- hnf(N,EN), EN = 1, hnf(Ys,[X0|_]), hnf(X0,H).
nth(N,Ys,H) :- hnf(Ys,[_|Xs]), hnf(N,EN), EN > 1, nth(EN-1,Xs,H).
f(X,H) :- hnf(1/(2+X*X),H).
```

The function *iterate* is translated as the following fact, where $@(F, X)$ is the FO representation for the expression $(F\ X)$ which appears in the definition.

```
iterate(F, X, [X | iterate(F, @(F,X))]).
```

The goal $nth\ 3\ (iterate\ f\ 0)\ ==\ X$ is translated into

```
nth(3, iterate(f, 0), H), `==_hnf´(H, X).
```

Solving the goal would partially evaluate `iterate(f,0)` to the list:

```
[0, @(f,0), @(f, @(f,0)) | iterate(f, @(f, @(f, @(f,0))))]
```

Only its third element would be evaluated for returning the result 0.444444 bound to the variable H and then, to the variable X, what would yield the solution $X = 0.444444$ for the initial goal.

Another possible optimization is related to **sharing**. On the implementation level, lazy narrowing should avoid the repeated evaluation of multiple occurrences of an expression which has been *passed as actual parameter*, i.e., introduced by the application of some rule whose righthand side is not linear. In our example of Sect. 3.2, where for instance terms of the form $(f^n(x_0), f^n(x_0) - f^{n-1}(x_0))$ are handled, sharing turns to be essential for efficiency. We have incorporated to the ETC translation the technique introduced by Cheong for implementing sharing[3] by means of Prolog variables.

There are still many sources for optimizing the translation. For instance, the occur check scheme performs redundant work in many occasions. A more efficient version similar to that in [1, 17] could be implemented.

5 Conclusions

We have presented $CFLP(\mathcal{R})$, a functional logic programming language enhanced with the possibility of using real arithmetic constraints. By means of examples we have shown the interest of the language for a wide range of problems which include all $CLP(\mathcal{R})$ applications and typical uses of functional programming for numerical algorithms.

For specifying the operational semantics, which incorporates real constraint solving to lazy narrowing, we have followed a quite simple idea: to rely on the constraint solving mechanism of a CLP system, and to integrate it into a previous scheme for translating lazy narrowing into Prolog. With respect to narrowing strategies we have considered, for the presentation of the work, the simplest strategy for lazy narrowing studied in [15], but the approach seems clearly extensible to other strategies.

We have in mind several natural ways for continuing our work: to complete the implementation, since there are still room for many optimizations; to consider other lazy narrowing strategies; to incorporate other kinds of constraints like disequality constraints for syntactical terms [1] or constraints over finite domains.

[3] Due to space limitations we omit the needed modifications which can be found in [2].

References

[1] 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.

[2] Cheong P.H., Fribourg L.: *Implementation of Narrowing: The Prolog-Based Approach.* In Apt K.R., de Bakker J.W., Rutten J.J.M.M. (eds) *Logic programming languages: constraints, functions, and objects*, The MIT Press, 1–20, 1993.

[3] ECL^{i}PS^e 3.5 ECRC Common Logic Programming System: *Extensions User Manual* and *User Manual.* December 1995, European Computer-Industry Research Center GmbH, Munich, Germany.

[4] Frankenstein H.: *Erweiterung von BABEL um lineare Constraints über reellen Zahlen.* Diplomarbeit, Lehrstuhl für Informatik II, Aachen Univ., April 1995.

[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] Hanus M.: *Efficient Translation of Lazy Functional Logic Programs into Prolog.* Procs. of LOPSTR'95, Springer LNCS 1048, 252–266.

[8] 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.

[9] Holzbaur C.: *OFAI clp(Q,R) Manual.* Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.

[10] Hughes J.: *Why Functional Programming Matters.* The Computer Journal 32 (2), 98–107, 1989. Also in D. Turner (ed) *Research Topics in Functional Programming*, Addison Wesley, 17–42, 1990.

[11] Jaffar J., Lassez J.L.: *Constraint Logic Programming.* Procs. of the 14th ACM Symp. on Principles of Programming Languages, 114–119, Munich 1987.

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

[13] Jaffar J., Michaylov S., Stuckey P.J., Yap R.H.C.: *The CLP($\mathcal{R}$) Language and System.* ACM Transactions on Programming Languages and Systems, Vol. 14, No. 3, 339–395, July 1992.

[14] Jiménez-Martín J.A., Mariño-Carballo J., Moreno-Navarro J.J: *Efficient Compilation of Lazy Narrowing into Prolog.* Procs. of LOPSTR'92, Springer Workshops in Computing Series, 253–270, 1992.

[15] 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.

[16] López-Fraguas F.J.: *A General Scheme for Constraint Functional Logic Programming.* Procs. of ALP'92, Springer LNCS 632, 213–217, 1992.

[17] López-Fraguas F.J.: *Programación funcional y lógica con restricciones.* PhD thesis, DIA-UCM, Madrid 1994.

[18] Moreno-Navarro J.J., Rodríguez-Artalejo M.: *Logic Programming with Functions and Predicates: The Language BABEL.* Journal of Logic Programming 12, 189–223, 1992.