

Functional Logic Programming with Real Numbers*

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

Dpto. Informática y Automática, Fac. Matemáticas, (UCM), Madrid, (SPAIN), E-28040
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. The execution mechanism of the language consists of a combination of lazy narrowing and constraint solving. 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 [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, 17], 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 [11] for a survey). To link together these two independent branches of evolution of logic programming appears as a natural interesting task. In [20, 21] 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., [22]) in the same way that $CLP(\mathcal{X})$ extended traditional logic programming. There are other proposals [3, 23] for CFLP (see [21] for a comparison) 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 [21] for a thorough exposition of that language and the $CFLP(\mathcal{X})$ scheme). We have recently known of a work [5], still in quite an early stage of development, where the functional logic language BABEL [22] 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 [8, 7]. Two *a priori* reasons encouraged us to start our work: the success of $CLP(\mathcal{R})$ and well-known classical examples [13] showing the expressiveness of lazy functional languages for programming numerical algorithms.

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

The rest of the paper is organized as follows: in Sect. 2 we describe the language and its syntax together with an example that demonstrates the usefulness of combining lazy functions with constraint solving over real numbers, exploiting lazy evaluation over infinite data structures. In Sect. 3 we explain the execution mechanism of the language, by means of a set of rules for goal transformation. Section 4 is devoted to the implementation of the language, realized through a translation of $CFLP(\mathcal{R})$ -programs into a constraint logic programming language. Section 5 includes a short set of experimental results attempting to show, if not the efficiency, at least the suitability of the approach. Finally, Sect. 6 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 [8, 7], 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. Consider the following sets of symbols and syntactic constructions:

- $tc, tc_1, \dots \in TCS$: *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 $\alpha, \beta, \dots \in TVar$, determine the set of *polymorphic types* $\tau, \tau_1, \dots \in Type$ defined by $\tau ::= \alpha \mid tc/0 \mid (tc/n \tau_1 \dots \tau_n) \mid (\tau_1, \dots, \tau_n) \mid (\tau_1 \rightarrow \tau_2)$. We assume, as usual, that \rightarrow is right associative. The first four alternatives in the previous definition determine the set $CType$ of *constructed types*.
- $c, c_1, \dots \in CS$: *data constructor* symbols, each with associated type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ ($\tau, \tau_i \in CType$). n is the arity of the constructor symbol. We assume that CS includes $[]/0, [\cdot]/2$ (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 $real \rightarrow real \rightarrow real$ (except $-/1$, which has type $real \rightarrow real$). In addition, we consider a set $Real$ of suitable representations (floating point, for instance) of (a set of) real numbers. Elements a, b, \dots of $Real$ have type $real$.
- $f, g, \dots \in FS$: *user defined function* symbols, each with associated *program arity* n and type $\tau_1 \rightarrow \dots \rightarrow \tau_m \rightarrow \tau$ ($\tau, \tau_i \in Type$, τ does not contain \rightarrow). 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 $X, Y, \dots \in Var$ of data variables, the set of *expressions* $e, l, r, \dots \in Exp$ is defined by $e ::= X \mid a \mid c \mid f \mid (e_1, \dots, e_n) \mid (e_1 e_2)$, 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 (see, e.g. [24]). An expression is *primitive* if no defined function appears in it. An important class of expressions is that of (first order) *patterns* $t, s \in Pat$, defined by $t ::= X \mid a \mid c t_1 \dots t_n \mid (t_1, \dots, t_n)$, 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 $\{ < /2, > /2, = < /2, > = /2, = \setminus = /2 \}$. Equality $==$ must be interpreted as *strict equality* (see e.g [22]).
- An *atomic constraint* $at \in Atom$ takes the form $e_1 \diamond e_2$ where $\diamond \in \Pi$. e_1, e_2 must be of type $real$ if $\diamond \in RRS$, and simply of the same (first order) type if \diamond is $==$. A *constraint* $\varphi, \psi, \dots \in Con$ is a multiset of atomic constraints, written in the form at_1, \dots, 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 $f t_1 \dots t_n = e \Leftarrow \varphi$, where $t_i \in Pat, e \in Exp, \varphi \in Con$. $t_1 \dots t_n$ must be linear, i.e., no variable can occur more than once. Each rule has a conditional reading: the value

of $f t_1 \dots t_n$ (the head) is the value of ϵ (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 [21] for an abstract semantic notion of nonambiguity which can be strengthened to decidable conditions similar to that in [22]. 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 $p(t_1, \dots, t_n) :- b_1, \dots, b_m, \varphi$ and can be straightforwardly¹ translated into our language as rules $p t_1 \dots t_n = true \Leftarrow b_1 == true, \dots, b_m == true, \varphi$. With this translation in mind, any $CLP(\mathcal{R})$ -program can be seen as a $CFLP(\mathcal{R})$ -program, thus determining a wide range of applications for our language.

- A *goal* is simply a constraint φ , whose variables have an existential reading. Computed answers 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. 3, 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 [8, 6, 7], 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 [6, 7] for details). The approach allows the use of HO logic variables in programs and goals. The operational semantics described in Sect. 3 and the implementation in Sect. 4 assume this FO translation of programs.

To give a complete semantic characterization of $CFLP(\mathcal{R})$ is far out of the scope of this paper. Some words in order to clarify the picture: in [21] 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. [21] also shows that the FO approach to HO functional logic programming of [6] 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.

2.1 An Example: “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, \dots is generated, starting from an arbitrarily chosen initial x_0 . The sequence is given by $x_0, x_1 = f(x_0), x_2 = f(x_1), x_3 = f(x_2), \dots$. 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 $[x_0, x_1, x_2, \dots]$ we can use the following polymorphic function, which is standard in lazy functional programming:

$$\begin{aligned} \text{iterate} &:: (A \rightarrow A) \rightarrow A \rightarrow [A] \\ \text{iterate } F \ X &= [X | \text{iterate } F \ (F \ X)] \end{aligned}$$

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

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

```

delta :: [real] → [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* $[x_0, x_1, x_2, \dots] = [(x_1, x_1 - x_0), (x_2, x_2 - x_1), \dots]$. Another standard list processing function (*zip*) is used in the definition:

```

accurate_sequence :: [real] → [(real, real)]
accurate_sequence [X|Xs] = zip Xs (delta [X|Xs])

zip :: [A] → [B] → [(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 $[(x_1, \epsilon_1), (x_2, \epsilon_2), \dots]$ of successive approximations paired with their corresponding error estimation.

```

accurated_iterations :: (real → real) → real → [(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 <- 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)

```

In our language, 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).

Our language allows to use non-ground goals (forbidden in functional programming). 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)           N = 2, Approx = (0.444444, -0.055555)
N = 3, Approx = (0.455056, 0.010611) N = 4, Approx = (0.453088, -0.001968)
...

```

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.

If we want to avoid lengthy goals we can still introduce, for a top level use of the program, a new function *nth_iter_step* which, for given f, x_0, n and ϵ returns x_n if the bound $|\epsilon_n| < \epsilon$ is satisfied.

```

nth_iter_step F X0 N Eps = XN
    <- nth N (accurated_iterations F X0) == (XN, E), E < Eps, -E < Eps

```

Notice that XN is a variable not occurring in the head of the rule. But since the condition *nth N (accurated_iterations F X0) == (XN, E)* implies a functional dependency of XN with

respect to $N, F, X0$, the use of XN in the righthand side is allowed². With this function our last goal could be reformulated as $nth_iter_step\ f\ 0\ N\ 0.01 == XN$.

Some final comments about higher order functions. Due to our treatment of HO functions, nothing avoids to consider goals with HO variables. The answers for these goals would include bindings of those HO variables. For instance, we could propose the goal $nth_iter_step\ F\ 0\ 2\ 0.05 == X$. An answer to this goal would be $F = g, X = 0.219824$. We have not yet explored the practical possibilities of HO logic variables in the specific context of functional logic programming with real numbers. Some examples showing the interest of HO functional logic programming can be found in [7]. Of course all those examples will work in our language, since it is an extension of the language in [7].

3 Operational Semantics of $CFLP(\mathcal{R})$ programs

As it is commented in Sect. 2, $CFLP(\mathcal{R})$ deals with higher order programming by translating HO expressions into first order [8, 6, 7]. Therefore, we will present the operational semantics of $CFLP(\mathcal{R})$ programs through their associated first order translations.

We assume a constraint solving mechanism for real primitive constraints: $\varphi \hookrightarrow_{\sigma}^{CS} \psi$, where φ, ψ are real primitive constraints, and σ is a substitution collecting the bindings for the variables in φ that have been possibly produced while solving it. We say that a real primitive constraint is in solved form if it is \hookrightarrow^{CS} -irreducible. We recall that initial goals are constraints φ involving the relational symbols in Π . While goal solving proceeds, some unification conditions will appear when applying a rule $f(t_1, \dots, t_n) = e \Leftarrow \varphi$ of the program for reducing an expression $f(e_1, \dots, e_n)$. We will write these unification problems as $e_1 = t_1, \dots, e_n = t_n$. Therefore, while describing the goal solving mechanism, we must consider intermediate goals of the form $U \otimes \varphi$, where U is a sequence of unification conditions and φ is a constraint. Now, we define a *one-step of goal solving* relation $U_1 \otimes \varphi \hookrightarrow_{\sigma} U_2 \otimes \varphi'$, where σ is a substitution (ε denotes the empty substitution). We say that $\langle \theta, \varphi_n \rangle$ is an *answer* for a goal φ_0 if there exists a finite sequence of \hookrightarrow -step:

$$\otimes \varphi_0 \hookrightarrow_{\sigma_0} U_1 \otimes \varphi_1 \hookrightarrow_{\sigma_1} \dots \hookrightarrow_{\sigma_{n-1}} \otimes \varphi_n$$

such that $\theta = \sigma_0 \dots \sigma_{n-1}|_{var(\varphi_0)}$, and φ_n is a real primitive constraint in solved form.

In the following we assume some well defined way of selecting positions in a constraint or in a sequence of =-equations. Given a constraint φ , by $[\varphi]_u$ and $\varphi[u \leftarrow e]$ we mean the subexpression of φ at position u , and the replacement in φ of $[\varphi]_u$ by e , respectively. The same can be applied for sequences of =-equations. Given $U \otimes \varphi$, $(U \otimes \varphi)[u \leftarrow e]$ means $U[u \leftarrow e]$ if u is a position belonging to U , or $\varphi[u \leftarrow e]$ otherwise. See now the rules associated to \hookrightarrow :

- **Constraint Solving:** $U \otimes \varphi, \psi \hookrightarrow_{\sigma} (U \otimes \varphi', \psi)\sigma$, if φ is a real primitive constraint and $\varphi \hookrightarrow_{\sigma}^{CS} \varphi'$.
- **Narrowing:** $U \otimes \varphi \hookrightarrow_{\varepsilon} (U \otimes \varphi)[u \leftarrow e] \cup e_1 = t_1, \dots, e_n = t_n, \psi$, if $[U \otimes \varphi]_u = f(e_1, \dots, e_n)$, $f \in FS^n$, u is a demanded position in $U \otimes \varphi$, and $f(t_1, \dots, t_n) = e \Leftarrow \psi$ is a variant (with fresh variables) of a rule of a $CFLP(\mathcal{R})$ -program P .

The intended meaning for the use of \cup is to put the unification conditions $e_i = t_i$ and ψ in the lefthand side and righthand side of \otimes respectively.

For the notion of demanded position, we can adopt the following syntactic criterion: a position u occupied by a function symbol (a *function position*, in short) is demanded in $U \otimes \varphi$ if it is an outer function position (i.e., not prefixed by any other function position) in φ or it is the root position of l in a component of U of the form $l = c(t_1, \dots, t_n)$.

These conditions are quite conservative, in the sense that they do not detect some positions that are “semantically” demanded. For dealing with sophisticated narrowing strategies like those presented in [19], demandedness conditions should be stressed. Nevertheless, the conditions above are powerful enough for ensuring that at least one of the function positions in a goal such that none of the rest of the rules for \hookrightarrow is applicable, is a demanded one.

²In this way we achieve an effect similar to the use (but in a quite restricted way) of **where** constructions in typical functional languages.

• **Rules for unification:**

(Decomposition) $c(e_1, \dots, e_n) = c(t_1, \dots, t_n), U \otimes \varphi \hookrightarrow_\varepsilon e_1 = t_1, \dots, e_n = t_n, U \otimes \varphi$, where $c \in CS^n$;

(Input Binding) $e = X, U \otimes \varphi \hookrightarrow_\sigma (U \otimes \varphi)\sigma$, where $\sigma = \{X/e\}$. This rule expresses parameter passing. Notice that “sharing” is not captured by this rule. As will be indicated in next section, the implementation can take care of sharing by using some well-known techniques [2].

(Output Binding) $X = t, U \otimes \varphi \hookrightarrow_\sigma (U \otimes \varphi)\sigma$, where $\sigma = \{X/t\}$.

• **Rules for equation solving:**

(Decomposition) $U \otimes c(e_1, \dots, e_n) == c(e'_1, \dots, e'_n), \varphi \hookrightarrow_\varepsilon U \otimes e_1 == e'_1, \dots, e_n == e'_n, \varphi$, where $c \in CS^n$;

(Deletion) $U \otimes X == X, \varphi \hookrightarrow_\varepsilon U \otimes \varphi$;

(Binding) $U \otimes X == t, \varphi \hookrightarrow_\sigma (U \otimes \varphi)\sigma$, if t is primitive, $t \neq X$ and $X \notin \text{var}(t)$, where $\sigma = \{X/t\}$;

(Imitation) $U \otimes X == c(e_1, \dots, e_n), \varphi \hookrightarrow_\sigma (U \otimes X_1 == e_1, \dots, X_n == e_n, \varphi)\sigma$, where $\sigma = \{X/c(X_1, \dots, X_n)\}$, X_i are fresh variables, $c \in CS^n$ but $c(e_1, \dots, e_n)$ is non-primitive.

In order to guarantee that this set of rules works properly (in particular for ensuring that unification is free of occur check), we must assume that whenever a narrowing step introduces $e_1 = t_1, \dots, e_n = t_n, \varphi$ as new conditions, the unification part $e_1 = t_1, \dots, e_n = t_n$ is solved before attempting to solve the constraint φ . This is not a severe limitation in practice, since it is in fact the usual way of proceeding. A more sophisticated set of rules (similar to those in [9, 21], which also capture sharing) could be adopted in order to allow a more flexible use, at the price of imposing additional, harder to implement, restrictions in the application of some of the rules.

4 Implementation

The implementation is conceived as a compilation to the CLP system over rationals or reals $\text{clp}(\mathbb{Q}, \mathbb{R})$ [12] running on a Prolog system extended with attributed variables (or metaterms). We have chosen ECL^iPS^e 3.5.2 [4] as the Prolog extension based on attributed variables³ on which $\text{clp}(\mathbb{Q}, \mathbb{R})$ is running. In the rest of the paper, we call this combination $\text{ECLP}(\mathbb{R})$. The $\text{clp}(\mathbb{Q}, \mathbb{R})$ 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).

Any given $\text{CFLP}(\mathbb{R})$ -program P is implemented as a set of clauses $\text{ETC}(P)$. 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, 10, 18, 19]. More concretely, our work is based on [19], where the reader can find more details about the translation into Prolog of different strategies, and where the *demand driven* strategy (*dds* for short) is presented. When introducing the implementation we use the simplest one for the sake of clarity: the *naive* strategy. In Sect. 5 we will show some run time results for both strategies.

Within $\text{ETC}(P)$ we represent $\text{CFLP}(\mathbb{R})$ 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^iPS^e -unification and the $\text{clp}(\mathbb{Q}, \mathbb{R})$ constraint solving mechanism. $\text{ETC}(P)$ consists of clauses for two main predicates: *hnf*, which narrows expressions into *head normal form*, and *solve*, which solves constraints (of rules and goals).

³Attributed variables are a special ECL^iPS^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 [4].

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 :

$$\begin{aligned}
 hnf(E, H) & :- \text{var}(E), !, H = E. \\
 hnf(E, H) & :- \text{number}(E), !, H = E. \\
 hnf((E_1, \dots, E_m), H) & :- !, H = (E_1, \dots, E_m). \\
 hnf(c(E_1, \dots, E_m), H) & :- !, H = c(E_1, \dots, E_m). \quad \% \text{ for each } c \in CS^m \\
 hnf(f(E_1, \dots, E_n), H) & :- !, \#f(E_1, \dots, E_n, H). \quad \% \text{ for each } f \in FS^n \\
 hnf(\diamond E, H) & :- !, hnf(E, E'), \{H = \diamond E'\}. \quad \% \text{ for each } \diamond \in RFS^1 \\
 hnf(E_1 \diamond E_2, H) & :- !, hnf(E_1, E'_1), hnf(E_2, E'_2), \{H = E'_1 \diamond E'_2\}. \quad \% \text{ for each } \diamond \in RFS^2
 \end{aligned}$$

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 $f(t_1, \dots, t_n) = e \Leftarrow \varphi$ of a $CFLP(\mathcal{R})$ -program P , we define an associated ECLP(\mathcal{R}) clause:

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

We use $\text{unify}(E, T)$ 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 . The predicate solve corresponds to the constraint solver and it is defined in the next subsection.

Clauses for unify :

$$\begin{aligned}
 \text{unify}(E, X) & :- \text{var}(X), !, X = E. \\
 \text{unify}(E, X) & :- \text{number}(X), !, hnf(E, E'), \{E' = X\}. \\
 \text{unify}(E, (T_1, \dots, T_m)) & :- !, hnf(E, (E_1, \dots, E_m)), \text{unify}(E_1, T_1), \dots, \text{unify}(E_m, T_m). \\
 \text{unify}(E, c(T_1, \dots, T_m)) & :- !, hnf(E, c(E_1, \dots, E_m)), \text{unify}(E_1, T_1), \dots, \text{unify}(E_m, T_m). \\
 & \quad \% \text{ for each } c \in CS^m \ (m \geq 0)
 \end{aligned}$$

4.3 Goal Solving with Incremental Occur Check

The constraint solver solve handles the constraints of rules and goals. In clause $\text{solve}(L == R)$, which depends on the predicate $==_hnf(L, R)$, the strict equation between expressions L and R is solved by lazy narrowing, following [19]. For the rest of the clauses $\text{solve}(L \diamond R)$, the constraint $L \diamond R$ is managed by first evaluating L and R to *head normal form* L' and R' respectively and then letting the $\text{clp}(\mathcal{Q}, \mathcal{R})$ system to solve the constraint $\{L' \diamond R'\}$.

Clauses for solve :

$$\begin{aligned}
 \text{solve}((\varphi, \varphi')) & :- !, \text{solve}(\varphi), \text{solve}(\varphi'). \\
 \text{solve}(L \diamond R) & :- !, hnf(L, L'), hnf(R, R'), \diamond_hnf(L', R'). \\
 & \quad \% \text{ for each } \diamond \in \{==, >=, =<, =\backslash=, <, >\}
 \end{aligned}$$

Clauses for \diamond_hnf :

$$\begin{aligned}
 \diamond_hnf(L, R) & :- \{L \diamond R\}. \quad \% \text{ for each } \diamond \in \{>=, =<, =\backslash=, <, >\} \\
 ==_hnf(X, Y) & :- (\text{is_real_hnf}(X); \text{is_real_hnf}(Y)), !, \{X = Y\}. \\
 ==_hnf(X, H) & :- \text{var}(X), !, \text{bind}(X, H). \\
 ==_hnf(H, X) & :- \text{var}(X), !, \text{bind}(X, H). \\
 ==_hnf((L_1, \dots, L_m), (R_1, \dots, R_m)) & :- !, \text{solve}(L_1 == R_1), \dots, \text{solve}(L_m == R_m). \\
 ==_hnf(c(L_1, \dots, L_m), c(R_1, \dots, R_m)) & :- !, \text{solve}(L_1 == R_1), \dots, \text{solve}(L_m == R_m). \\
 & \quad \% \text{ for each } c \in CS^m \ (m \geq 0)
 \end{aligned}$$

Clauses for *is_real_hnf*:

The first clause of `==_hnf` uses the predicate *is_real_hnf* which is going to be defined next. Its intended use is to distinguish *head normal forms* standing for admissible $\text{clp}(\mathbb{Q}, \mathbb{R})$ 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^iPS^e by means of the type-checking predicate `meta/1`.

```
is_real_hnf(X) :- number(X),!.
is_real_hnf(X) :- 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) :- is_real_hnf(H),!,{X=H}.
bind(X,H) :- var(H),!,X=H.
bind(X,(E1,...,Em)) :- !,occurs_not(X,E1),...,occurs_not(X,Em),X=(X1,...,Xm),
                        hnf(E1,H1),bind(X1,H1),...,hnf(Em,Hm),bind(Xm,Hm).
bind(X,c(E1,...,Em)) :- !,occurs_not(X,E1),...,occurs_not(X,Em),X=c(X1,...,Xm),
                        hnf(E1,H1),bind(X1,H1),...,hnf(Em,Hm),bind(Xm,Hm).
                        % for each c in CS^m (m >= 0)

occurs_not(X,Y) :- var(Y),!,X\==Y. % X\==Y checks syntactic disequality.
occurs_not(X,c(E1,...,Em)) :- !,occurs_not(X,E1),...,occurs_not(X,Em).
                        % for each c in CS^m (m >= 0).

occurs_not(X,E).
```

5 Experimental Results

Before giving the experimental results, we comment briefly on several optimizations that we have introduced at the level of the *ETC* translation in order to compute the times showed in the table below.

First, we observe that calls to some predicates can easily be **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. 2.1 are:

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

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 right-hand side is not linear. In our example of Sect. 2.1, 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 sharing⁴ 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, 21] could be implemented.

⁴Due to space limitations we omit the needed modifications which can be found in [2].

For *dds* and similar strategies, there are some additional known optimizations [10] that may be successfully applied as redundant constructor elimination or argument swapping for better indexing. In our particular case, some possible optimizations are related to the constraint management. After several optimization steps, the clauses for *nth* in the *dds*-translation are:

```
nth(N,Ys,H)          :- hnf(Ys,[X0|Xs]), hnf(N,EN), nth_2_1(EN,X0,Xs,H).
nth_2_1(EN,X0,Xs,H) :- ({EN = 1} -> hnf(X0,H) ; ({EN > 1}, nth(EN-1,Xs,H))).
```

Finally, in order to write the program (given in Sect. 2.1) in *ECLP(R)* avoiding the use of infinite lists, we have used the following ECLP(R) program for comparing the run times.

```
nth_iterate(_,X,N,X) :- {N = 0}.
nth_iterate(F,X,N,Sol) :- {N > 0}, function(F), E =.. [F,X,X1], call(E),
    {N1 = N - 1}, nth_iterate(F, X1, N1, Sol).
nth_iter_step(F,X,N,Eps,Sol1) :- {N = 1}, function(F), T =.. [F,X,Sol1],
    call(T), {E = Sol1 - X, E < Eps, -E < Eps}.
nth_iter_step(F,X,N,Eps,Sol1) :- {N > 1}, {N1 = N - 1},
    nth_iterate(F,X,N1,Sol2), T =.. [F,Sol2,Sol1],
    call(T), {E = Sol1 - Sol2, E < Eps, -E < Eps}.
f(X,H) :- {H = 1 / (2 + X * X)}.
g(X,H) :- {H = (2 + X) / (10 + X * X * X)}.
function(f).
function(g).
```

The next table shows run time results obtained using the *dds* and naive strategies (in the terminology of [19]) applied to the program at hand, and the latter ECLP(R) version (with the required modification in the goals form). The run times for computing the first solution are in seconds. The program has been executed in a SPARC station 1+.

Goal	dds	nai	ECLP(R)	Result
<i>F1</i>	0.7	0.7	0.5	$N = 13.0, X_N = 0.453397$
<i>G1</i>	0.6	0.6	0.3	$N = 10.0, X_N = 0.221952$
<i>F2</i>	2.3	2.4	1.3	$X_N = 0.453397, \{Eps > 0.0\}$
<i>G2</i>	3.0	3.2	1.5	$X_N = 0.221952, \{Eps > 0.0\}$

where:

```
F1 ≡ nth_iter_step f 0 N 0.000000001 ==  $X_N$ .   G1 ≡ nth_iter_step g 0 N 0.000000001 ==  $X_N$ .
F2 ≡ nth_iter_step f 0 300 Eps ==  $X_N$ .       G2 ≡ nth_iter_step g 0 300 Eps ==  $X_N$ .
```

Obviously, all the CFLP(R) features (infinite data, functions, lazy evaluations) that are not available in ECLP(R) introduce some overhead, which nevertheless is not very high since for the worst cases the run times are increased by a factor of 2.

6 Conclusions

We have presented *CFLP(R)*, a functional logic programming language enhanced with the possibility of using real arithmetic constraints. Moreover, *CFLP(R)* includes all *CLP(R)* applications and typical uses of functional programming for numerical algorithms.

For implementing 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. The overhead in which the translation incurs, when compared with programs written directly in the object language ($ECL^iPS^e + clp(Q,R)$), can be considered as acceptable, taking into account the simplicity of the obtained implementation. With respect to narrowing strategies we have considered, for the presentation of the work, the simplest strategy for lazy narrowing studied in [19], but the approach seems clearly extensible to other strategies (one of which has been used in fact for the experimental results).

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 and to sophisticate the interleaving of constraint solving and lazy narrowing; to incorporate other kinds of constraints like disequality constraints for syntactical terms [1] or constraints over finite domains. In a different direction, we want to complete also the semantic characterization of the language, in the setting of the $CFLP(\mathcal{X})$ scheme.

References

- [1] Arenas-Sánchez P., Gil-Luezas A., López-Fraguas F.J.: *Combining Lazy Narrowing with Disequality Constraints*. Procs. of Int. Symp. on Programming Language Implementation and Logic Programming (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] 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.
- [4] ECLⁱPS^e 3.5 ECRC Common Logic Programming System: *Extensions User Manual and User Manual*. December 1995, European Computer-Industry Research Center GmbH, Munich, Germany.
- [5] Frankenstein H.: *Erweiterung von BABEL um lineare Constraints über reellen Zahlen*. Diplomarbeit, Lehrstuhl für Informatik II, Aachen Univ., April 1995.
- [6] González-Moreno J.C.: *A Correctness Proof for Warren's HO into FO Translation*. Procs. of GULP'93, 569–585, 1993.
- [7] González-Moreno J.C.: *Programación Lógico Funcional de Orden Superior con Combinadores*. PhD thesis, DIA-UCM, Madrid 1994.
- [8] 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 Computer Science Logic (CSL'92), Springer LNCS 702, 216–230, 1993.
- [9] 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.
- [10] Hanus M.: *Efficient Translation of Lazy Functional Logic Programs into Prolog*. Procs. of LOPSTR'95, Springer LNCS 1048, 252–266.
- [11] 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.
- [12] Holzbaur C.: *OFAL clp(Q,R) Manual*. Edition 1.3.3, Austrian Research Institute for Artificial Intelligence, Vienna, TR-95-09, 1995.
- [13] 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.
- [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.: *Methodology and Implementation of a Constraint Logic Programming System*. Procs. of the 4th Int. Conf. on Logic Programming (ICLP'87), MIT Press, 196–218, 1987.
- [17] 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.
- [18] 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.
- [19] Loogen R., López-Fraguas F.J., Rodríguez-Artalejo M.: *A Demand Driven Computation Strategy for Lazy Narrowing*. Procs. of Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93), Springer LNCS 714, 184–200, 1993.
- [20] López-Fraguas F.J.: *A General Scheme for Constraint Functional Logic Programming*. Procs. of Int. Conf. on Algebraic and Logic Programming (ALP'92), Springer LNCS 632, 213–217, 1992.
- [21] López-Fraguas F.J.: *Programación funcional y lógica con restricciones*. PhD thesis, DIA-UCM, Madrid 1994.
- [22] Moreno-Navarro J.J., Rodríguez-Artalejo M.: *Logic Programming with Functions and Predicates: The Language BABEL*. Journal of Logic Programming 12, 189–223, 1992.
- [23] Mück A., Streicher T., Lock H.: *A Tiny Constraint Functional Logic Language and Its Continuation Semantics*. Procs. of European Symp. on Programming ESOP'94, Springer LNCS 788, 439–453, 1994.
- [24] Peyton Jones S.L.: *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

