DL Prolog: Another Unifying Programming Language

José Oscar Olmedo-Aguirre

Departamento de Ingeniería Eléctrica, Cinvestav, Ciudad de México, D.F., México
oolmedo@cinvestav.mx

Paper received on 30/07/12, Accepted on 16/08/12.

Abstract. Programming language research has been guided by the hope of identifying a minimal set of expressive enough programming concepts that can describe uniformly and consistently most computational problems along with their solutions. DL Prolog unifies the logic, functional and imperative programming paradigms by the introduction of dynamic logic modalities into pure Prolog with equational unification. The imperative fragment of DL Prolog uses equations between terms to define logical variables and a novel assignment to redefine imperative variables. The contribution of this paper is in introducing these two operations in the context of the imperative fragment of DL Prolog that are formally described in the structured operational semantics style. A working research prototype has been built upon this design and it is available on request from the author.

Keywords: Programming Language, Logic Programming, Dynamic Logic, Imperative Programming, Unification, Prolog**.**

1 Introduction.

Unified programming research attempts to identify the minimal set of the simplest and most fundamental programming concepts that coherently can be applied in as many as diverse areas of application and still can lead to reasonable efficient implementations. Among the most successful efforts in conciliating such a diversity of concepts, the functional logic programming paradigm is capable of embedding the notion of state that characterizes the imperative programming, along with constraint satisfaction, concurrent processing and object orientation, among others. Nonetheless, there is neither symmetry nor balance in their integration in the sense that a Java programmer, for example, needs to learn a number of sophisticated concepts that are uncommon in her experience to start programming even relatively simple programs. In this respect, an extension to pure Prolog with dynamic logic modalities, named DL Prolog [8], has been designed by the author to provide a more balanced approach in the sense that the same Java programmer has not to learn too many concepts if she does not need to. Instead of placing the imperative programming at the top of the functional logic programming paradigm as in Curry [4], the unified approach of DL Prolog stands upon the imperative, functional and logic programming paradigms. Unfortunately, due to lack of space, the previous claim will only be justified by discussing some program examples that will also help to show the programming style of the imperative fragment of DL Prolog.

This article is organized as follows. In section 2, a succinct review of the related work is presented. In section 3, the programming examples describe the use of unification. In section 4, a brief account of the programming style of the imperative fragment of DL Prolog is shown. In section 5, a formal description in the structured operational semantics is given for the imperative fragment. Finally in section 6, some further research directions and concluding remarks are given.

2 Related work.

Classical papers on the integration of functional, equational and logic programming were collected in [1]. Among them, Kahn's Uniform language is the closest to ours in its use of unification though his proposal was developed for the functional language LISP, whereas DL Prolog applies unification for its imperative fragment [2]. Uniform is an AI language that uses augmented unification to solve equational problems that cannot be solved by syntactic unification. Probably the most serious difficulty faced in the design of Uniform was the computational complexity of finding a solution in presence of equality theories that lead to large chains of equalities. In fact, solving equational problems in presence of equality theories is known to be a NP complete problem [3]. In DL Prolog the combinatorial production of equalities has been drastically reduced by (i) incorporating a greater degree of control into the language by means of functions instead of equations, and (ii) avoiding comparing two functional terms, so any equation can relate at least one constructor-rooted term. Thus DL Prolog unification prevents applying higher-order unification which is a NP-complete problem [3].

As discussed by Kahn, Uniform lacked of more theoretical foundations in equational unification to deal with the combinatorial search involved in the generation of solutions. Taking advantage of the reasonably efficient implementations of functional languages, the integration efforts were later oriented to approach relations as Boolean functions. Curry [4] is a functional logic programming language based on the evaluation by need of expressions (a mechanism called *narrowing* used in lazy functional programming) along with the possible instantiation of free variables occurring in the expressions (a mechanism called *residuation* used in constraint-logic programming). Nonetheless, imperative programming is no directly available but through a monadic IO system that enable sequence of input-output actions. Despite its growing success and acceptance, the use of evaluation by need makes difficult to reason about program behavior and memory requirements. This means that a programmer needs to understand the not so intuitive IO monadic system in order to write even the simplest imperative programs.

Concurrent programming, particularly the so-called algebraic theory of processes has provided a basis for its integration with the functional, imperative and object-oriented paradigms. The language PICT based on the π-calculus is an example of this approach [5]. In addition to the formidable task of implementing enough programming concepts as networks of interacting processes, the several layers that will emerge from such design makes no clear how efficient and of practical use would be this integration. However, the concurrent programming approach has also leaded to more successful pragmatic language designs like Oz. Oz is termed as a multi-paradigm programming language supporting logic, functional, imperative, object-oriented, constraint, concurrent and distributed paradigms [6]. Oz is based on residuation, a simple yet powerful mechanism to solve concurrently constraints by processes that suspend their execution whenever a variable is undefined in a predicate and resume their execution soon after the variable becomes defined.

3 Unification.

The importance of unification and equational unification (E-unification) is due to the fact that is widely used in automated theorem proving and related fields like logic programming. The term unification generally stands for syntactic equality on terms, whereas semantic E-unification generally stands for syntactic equality modulo an equational theory. Augmented unification is a restricted version of E-unification to be used as a model of execution [2]. For a brief account of unification and E-unification, consider the following DL Prolog equations to define variables. Hereinafter, variables are typed in italics:

1. [*u*, b, c, d] = [a | *vs*]
2. (*us*, a, *x*, [*x* | *xs*]) = ([a, b], a, *y*, *us*)
3. f(*us*, [*us*]) = g([a, b], *vs*)
4. [*x*, *y* | *xs*] = rev([a, b])
5. [*x*, *y* | *xs*] = rev([a])

As term constructors are injections (like the list and the tuple constructors), solving the equation *t* = *s* amounts to finding an assignment of values to variables that makes terms *t* and *s* syntactically identical. Unification transforms the equation *t* = *s* into a set of consistent equations formed by comparing the corresponding subterms of *t* and *s* whenever they have the same root constructors and the same number of subterms. Thus, the equation (1) is transformed into the set of equations {*u* = a, [b, c, d] = *vs*}, where variables *u* and *vs* are respectively bound to a and [b, c, d]. Similarly the equation (2) is transformed into the set of equations {*us* = [a, b], a = a, *x* = *y*, [*x* | *xs*] = *us*}. From this set, the first equation defines the value of *us*, the second can be discarded, the third is recorded (as *x* and *y* remain unknown so far), and the last leads to the equations *x* = a and *xs* = [b]. After finding that *x* = a, variable *y* becomes also bound to a. In (3), the unification fails because the compared terms have different rooted constructors f and g, and there is not function definitions for either f or g (but not both) whose evaluation may result in a constructor-based term that could be used instead in the equation.

The last two examples deal with E-unification. In (4), sole unification cannot solve the equational problem because the constructor of the list is not equal to the function name rev that stands for the usual reverse function. As unification fails, E-unification is applied by searching for an equation or function defined for rev. If the search succeeds, the function is evaluated with the given parameters and its result is replaced where the function application occurs; otherwise, the E-unification fails and the equational problem has no solution. For this example, because there is a function definition with the usual meaning for rev, equation rev([a, b]) = [b, a] holds and equation (4) becomes [*x*, *y* | *xs*] = [b, a], producing the solutions *x* = b, *y* = a and *xs* = []. Finally, in (5) even E-unification fails because assuming that function rev is defined, the result of its application on [a] has only one element whereas the list [*x*, *y* | *xs*] can only unify with lists of at least two elements.

4 A DL Prolog Program Example.

The following DL program shows the dynamic logic modality introduced in the pure Prolog language and model. The program simply checks that the well-known property of lists rev(app(*xs*, *ys*)) = app(rev(*ys*), rev(*xs*)) holds for the lists [a, b, c, d] and [e, f, g, h].

```
[ new ys, zs: (
        xs = [a, b, c, d]; ys := [e, f, g, h];
        zs := rev(app(xs, ys)); us = zs; writeln(revapp : us);
        zs := app(rev(ys), rev(xs)); vs = zs; writeln(apprev : vs)
        )
] (us = vs).
```

This DL Prolog program consists of two parts: (i) the modal connective that encloses in brackets the well-formed imperative program: new ys, *zs*: (*xs* = [a, b, c, d], …), and (ii) the postcondition *us* = *vs* that states the properties of the values bound to the variables at the program termination. The property tested is valid in a theory of lists with the usual operations rev(*xs*) and app(*xs*, *ys*) that respectively reverses the order of the elements of list *xs* and appends the elements of list *ys* at the end of list *xs*. In the program, the *logical variable xs* is defined with list [a, b, c, d], whereas the *local imperative variable ys* is defined with list [e, f, g, h]. Logical variables can be defined by equality at most once, whereas local variables can arbitrarily be redefined as many times as needed by means of the destructive assignment of imperative programming. In any case, the occurrence of either a logical or local variable in an expression denotes the value bound to it. Program instructions are sequentially executed as usual from left to right of the semicolon connective. The assignment *zs* := rev(app(*xs*, *ys*)) defines the local variable *zs* with a list that results from reversing the concatenation of lists *xs* and *ys*. This list is used to define the logical variable *us* that is next written in the terminal output preceded by label revapp. The assignment *zs* := app(rev(*ys*), rev(*xs*)) redefines the local variable *zs* with the concatenation of the reversed lists of *ys* and *xs*. This list is used to define the logical variable *vs* that is next written preceded by label apprev. Beyond this point in the program text, the scope of the binder new reaches its end and the local variables *ys* and *zs* cease their existence. Nonetheless, logical variables *us* and *vs* remain unaltered at the program postcondition *us* = *vs*. The partial correctness of the reverse program ensures that the postcondition is always satisfied upon its termination whenever the input parameter satisfies a suitable precondition. Hence there is no need to test the postcondition as much as the partial correctness of the reverse program can be proved.

4.1 The reverse program

The DL Prolog program for the reverse function *rev*(*xs*) = *zs* is presented next in a stylized format that helps to show its syntactic structure:

$$\left[new\, u, us, vs : \begin{pmatrix} (us, vs) := ([\,], xs); \\ ([u \mid vs] := vs;\, us := [u \mid us])^*; \\ zs = us \end{pmatrix} \right] rev(xs) = zs :- list(xs).$$

The above program text is equivalent to its usual linear presentation [*new u*, *us*, *vs*: ((*us*, *vs*) := ([], *xs*); ([*u* | *vs*] := *vs*; *us* := [*u* | *us*]) *; *zs* = *us*?)] *rev*(*xs*) = *zs* :- *list*(*xs*) that can freely be formatted according to the programmer preferences. The above program is in fact the partial correctness property of the program to compute the reverse function. Like Prolog, the reverse program in DL Prolog is a Horn clause consisting of a simple conclusion predicate called the *head of the clause* and an antecedent formed by a conjunction of predicates called the *body of the clause*. Unlike Prolog, the head of the clause in DL Prolog is a *dynamic logic assertion* consisting of the reverse program enclosed in brackets followed by the program postcondition *rev*(*xs*) = *zs*. A DL Prolog program like reverse *fails*, either if it does not terminate, or if upon its completion, it cannot satisfy its postcondition.

The imperative programming constructs used in the DL Prolog assertions comprise the most common structured programming connectives for sequential, conditional and iterative composition. The program (A_1 ; A_2) stands for the sequential composition of the program (fragments) A_1 and A_2. The non-deterministic execution (A_1 :: A_2) stands for the non-deterministic selection of one of two program fragments A_1 and A_2: if both programs terminate, the non-deterministic composition terminates by arbitrarily choosing one; if both programs fail, the non-deterministic composition fails too; otherwise, if one program fails, the companying unfailing program is chosen. This behavior is required to introduce the usual conditional composition of programs (*C*?; *A*), where a possibly failing program fragment *C*, called *the guard*, prevents the execution to continue with program *A* if *C* is false. The usual if-then-else construct (if *C* then A_1 else A_2) is defined by the non-deterministic selection (*C*?; A_1 :: ¬*C*?; A_2) between two programs A_1 and A_2 guarded with mutually exclusive conditions *C* and ¬*C*. The unary suffix connective (*A**) stands for the iterative execution of *A*, where the repeated execution of *A* continues until it fails. Furthermore, the language is block structured, where local variables are introduced in the block by binder new. Local variables correspond to the variables of usual imperative languages that can be redefined by means of a newly assignment operator. The multiple-assignment operator extends the single assignment by assigning to a list of variables the values of a list of terms of the same length. Logical variables are mainly used to name values because they can be defined at most once, whereas local variables are used to store partial computations.

Though the previous program connectives have the usual meaning, the assignment has a remarkably unusual meaning as it is shown by analyzing the three assignments of the reverse program. In the assignment (*us*, *vs*) := ([], *xs*), local variables *us* and *vs* are initially defined by the empty list [] and the list *xs* to be reversed, respectively. In the assignment [*u* | *vs*] := *vs*, the list bound to *vs* is decomposed in a first element and in a list with all the elements of *vs* but the first. If *vs* contains at least one element, the decomposition is possible, and the assignment succeeds by binding the first element of *vs* to *u* and by removing the first element of the list bound to *vs*. However, if *vs* is the empty list, the decomposition of *vs* is not possible. Hence the assignment fails and so does the sequential composition in that it appears, causing the termination of the iteration. Finally, in the third assignment *us* := [*u* | *us*], the variable *us* is redefined by the list [*u* | *us*]. The term [*u* | *us*] denotes a list whose first element is *u* and the rest is *us*. Thus, after the assignment, the list *us* is as before augmented with the value of *u* placed at its first element. Note that among the three assignments, only [*u* | *vs*] := *vs* may fail due to the constraint that both lists at each side of the assignment must agree in their number of elements. In the context of the iterative connective in which this assignment occurs, this failing behavior is needed to ensure termination. At the end of the program fragment, the elements accumulated in list *us* define the value of the output variable *zs*.

4.2 The reverse postcondition

For the reverse program, the postcondition *rev*(*xs*) = *zs* asserts that the elements of *zs*, if any, are in reversed order with respect to those of *xs* whenever the precondition *list*(*xs*) holds. The postcondition of the reverse program establishes the formal relation between the input and the output variables, comprising three elementary conditions: (i) *xs* and *zs* must be both lists, (ii) the length of *xs* is equal to the length of *zs*, and (iii) when *xs* contains at least one element, the element of *xs* at position *i* is equal to the element of *zs* at position length(*xs*) – *i*, for any index *i* between 0 and length(*xs*) – 1. Fortunately, the partial correctness property of the program rev ensures that is not necessary to check that the postcondition holds at the end of each execution.

The modal assertion of DL Prolog extends logic programming with functions defined by imperative programs in a novel manner. Next the formal description of the language syntax and semantics is briefly presented, emphasizing the central role of E-unification in equations and assignments.

5. Formal Description.

The signature Σ,Ξ of DL Prolog comprises all its symbols grouped in two disjoint sets: the set Σ = {a, b, c, …, f, g,…} of constructor and function names and the set Ξ = {*u*, *v*, *x*, *y*, …} of variable names. Constants are constructors with no structure. The set *T*(Σ,Ξ) of terms with variables is the minimal set containing constants, variables and that is closed under the composition of constructors and functions from Σ with a (possibly empty) sequence of terms. The set *T*(Σ) of ground terms consists of terms with no variables. The following grammar rules describe the syntactic structure of the main categories of the language built upon the signature Σ,Ξ:

Terms $T(\Sigma,\Xi)$:

$t ::= x \mid \mathrm{c} \mid \mathrm{f}(t_1,\ldots,t_n)$

Predicates $P(\Sigma,\Xi)$:

$P ::= \bot \mid \top \mid t_1 = t_2 \mid \mathrm{p}(t_1,\ldots,t_n)$

Goals $G(\Sigma,\Xi)$:

$G ::= P \mid G_1 , G_2$

(Horn) Clauses $H(\Sigma,\Xi)$:

$H ::= P \mid P \text{ :- } G$

Programs $A(\Sigma,\Xi)$:

$A ::= t_1 = t_2 \mid t_1 := t_2 \mid C? \mid A_1 ; A_2 \mid A_1 :: A_2 \mid A^* \mid (A) \mid \mathrm{new}\ x_1,\ldots,x_n\text{: } A$

Modal programs $M(\Sigma,\Xi)$:

$M ::= P \mid [A]\, M$

Modal clauses $C(\Sigma,\Xi)$:

$C ::= M \mid M \text{ :- } G$

where the entities of each category are distinguished by indexing. The symbols $\bot$ and $\top$ stand for the truth values false and true, respectively. Horn clauses and modal clauses are assumed to be universally closed. The set of variable names fv(F) occurring free in a (modal) clause F occurring within the scope of a binder are those not captured by the binder, i.e. fv(new(x) F) = fv(F) – $\{x\}$ for the binder new and similarly for the universal and existential quantifiers.

Substitutions model program states. A *program state* is the set of bindings between variables and their bound values. Substitutions are partial functions $S(\Sigma,\Xi) = \Xi \rightarrow T(\Sigma,\Xi)$ from variables to terms. A *substitution* $\sigma : S(\Sigma,\Xi) = \{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ binds a set $\{x_1, ..., x_n\}$ of different variables, called the domain of the substitution dom(σ), with a set $\{t_1, ..., t_n\}$ of terms whenever x_i does not occur in t_i for $i = 1, \ldots, n$. A substitution is admissible if and only if none of the variables of t_i becomes bound by a binder after its substitution for all $i = 1, \ldots, n$. A *ground substitution* $\sigma : \Xi \rightarrow T(\Sigma)$ is a substitution on ground terms. A substitution σ is naturally extended as a map $T(\Sigma,\Xi) \rightarrow T(\Sigma,\Xi)$ from terms to terms and also for the other syntactic categories of the language. The application $t\sigma$ of a substitution σ on a term t (or the instance $t\sigma$ of t under σ) is a term obtained from the simultaneous replacement of x_i by t_i for $i = 1,\ldots,n$, such that:

- $\mathrm{c}\ \sigma = \mathrm{c}$, for all $\mathrm{c} \in \Sigma$
- $x\ \sigma = x$, for all $x \in \Xi$ and x not in dom(σ)
- $x_i\ \sigma = t_i$, for all x in dom(σ)
- $f(x_1,\ldots, x_n)\ \sigma = f(x_1\,\sigma,\ldots, x_n\,\sigma)$
- $[A]M\ \sigma = [\sigma][A]M$

where notation $[\sigma]$ stands for the conjunction of the equations $x_1 = t_1,\ldots, x_n = t_n$, also written as $(x_1,..., x_n) = (t_1,..., t_n)$. These equations are written in *solved form* whenever they are satisfied by the ground substitution $\sigma = \{x_1 \rightarrow t_1,\ldots, x_n \rightarrow t_n\}$. Note that unlike the application of a substitution to terms, the application to the program modality $[A]M\ \sigma$ is defined as the initial binding of the program variables $[\sigma][A]M$, i.e. not for the usual simultaneous replacement of all the variables of σ through the entire program A. The composition $\sigma_1\sigma_2$ of substitutions $\sigma_1 = \{x_1 \rightarrow t_1,\ldots, x_n \rightarrow t_n\}$ and $\sigma_2 = \{y_1 \rightarrow s_1,\ldots, y_m \rightarrow s_m\}$ is defined as:

$$\begin{aligned}\sigma_1\sigma_2 \quad &= \quad \{x_i \mapsto t_i\sigma_2 \mid x_i \in \mathrm{dom}(\sigma_1) \wedge t_i\sigma_2 \notin \mathrm{dom}(\sigma_1), i = 1,\ldots,n\}\\ &\cup \quad \{y_i \mapsto s_i \mid y_i \in \mathrm{dom}(\sigma_2) \wedge y_i \notin \mathrm{dom}(\sigma_1), i = 1,\ldots,m\}\end{aligned}$$

The unification of two terms either finds a unifier, i.e. a substitution that solves the equational problem or terminates in failure. Unification is a step by step transformation process between sets of equations until no further transformations can be applied. The unification procedure is defined through the following rules.

- $\left(\{t = t\} \cup P, S\right) \to \left(P, S\right)$
- $\left(\{f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)\} \cup P, S\right) \to \left(\{t_1 = s_1,\ldots,t_n = s_n\} \cup P, S\right)$
- $\left(\{f(t_1,\ldots,t_n) = g(s_1,\ldots,s_m)\} \cup P, S\right) \to \left(\{\},\{\perp\}\right)$
- $\left(\{x = t, x = s\} \cup P, S\right) \to \left(\{x = s, t = s\} \cup P, S\right)$ if $x = t \gtrsim t = s$ for some appropriate well-founded lexicographic ordering $\gtrsim$
- $\left(\{x = t\} \cup P, S\right) \to \left(\{\},\{\perp\}\right)$ if $x \in fv(t)$ and $x \neq t$
- $\left(\{x = t\} \cup P, S\right) \to \left(P\{x \mapsto t\}, S\{x \mapsto t\} \cup \{x = t\}\right)$ if $t \notin \Xi, x \notin fv(t)$

By applying the unification rules to the equational problem $t = s$, the pair $(\{t = s\},\{\})$ is transformed into the pair $(\{\}, S)$, meaning that the set S of equations in solved form is the solution to the problem; otherwise, the pair $(\{t = s\},\{\})$ is transformed into the pair $(\{\},\{\perp\})$, meaning that the equational problem has no solution.

The semantic description of DL Prolog is established according to the *structural operational semantics* [7] by defining program descriptions and transitions between program descriptions. Assuming the termination of the program, a *program description*, or simply a *description*, can be either instantaneous or final. An *instantaneous description* $I(\Sigma,\Xi) : M(\Sigma,\Xi)\times S(\Sigma,\Xi)$ is a pair $([A]M, \sigma)$ relating a program modality $[A]M$ and a program state σ, meaning that the program A starts its execution in the state σ, reaching a state that is assumed to satisfy M whenever A terminates. Note that M may be either another program modality or a predicate. For the former case, a new instantaneous description can be established, whereas for the latter case, a final description is reached. A *final description* $F(\Sigma,\Xi) : S(\Sigma,\Xi) \vee \{\perp\!\!\perp\}$ corresponds to the state reached when the program terminates or fails. The final description of a program that terminates successfully is a state $\sigma : S(\Sigma,\Xi) = \{x_1 \to t_1,\ldots, x_n \to t_n\}$, containing the bindings of the variables x_i with their final values t_i for $i = 1,\ldots, n$. Instead, the final description of a program that terminates in failure is represented by $\perp\!\!\perp$. A *transition* is a relation over pairs of program configurations $\to : I(\Sigma,\Xi) \times (I(\Sigma,\Xi) \vee F(\Sigma,\Xi))$. A *execution of a program* corresponds to the reflexive and transitive closure of the transition relation.

The execution of a terminating program is a finite sequence of instantaneous descriptions ended by a final description. For the successful program execution:

$(M_0, \sigma_0), (M_1, \sigma_1), \ldots, (M_i, \sigma_i), (M_{i+1}, \sigma_{i+1}), \ldots, (M_n, \sigma_n), \sigma_n$, for some $n \geq 0$,

the program starts with the initial modality $M_0 = [A]P$ along with the variables initialized according to σ_0 and the program terminates reaching the state σ_n, satisfying the partial correctness property. Thus, $[A]P\sigma_0$ implies $P\sigma_n$. This property follows by induction on the length n of the sequence of instantaneous descriptions, where $M_i\ \sigma_i$ implies $M_{i+1}\ \sigma_{i+1}$ for all $i = 1, \ldots, n$. For the unsuccessful program execution:

$(M_0, \sigma_0), (M_1, \sigma_1), \ldots, (M_i, \sigma_i), (M_{i+1}, \sigma_{i+1}), \ldots, (M_n, \sigma_n), \perp\!\perp$, for some $n \geq 0$,

the program starts like before, though it now terminates in failure, meaning that nothing can be asserted about the final program state.

The following inference rules define the SOS semantics of the imperative fragment of DL Prolog:

1. $\dfrac{(\{f(t_1,\ldots,t_n)\sigma = f(s_1,\ldots,s_n)\sigma\},\{\}) \rightarrow (\{\},\{x_1 = r_1,\ldots,x_m = r_m\})}{([f(t_1,\ldots,t_n) = f(s_1,\ldots,s_n)]M,\sigma) \rhd (M,\sigma\{x_1 \mapsto r_1,\ldots,x_m \mapsto r_m\})}$, with $0 \leq n$
2. $([f(t_1,\ldots,t_n) = g(s_1,\ldots,s_m)]M,\sigma) \rhd \perp\!\perp$ if f and g are different and there are not defined functions for either f or g, with $0 \leq m, n$
3. $\dfrac{(\{g(s_1\sigma,\ldots,s_m\sigma) = s, t\sigma = s\},\{\}) \rightarrow (\{\},\{x_1 = r_1,\ldots,x_n = r_n\})}{([t = g(s_1,\ldots,s_m)]M,\sigma) \rhd (M,\sigma\{x_1 = r_1,\ldots,x_n = r_n\})}$ if there is defined a function g such that $g(s_1\sigma,\ldots,s_m\sigma) = s$, with $0 \leq m, n$
4. $([t := s]M,\sigma_1\sigma_2) \rhd ([t = s\sigma_1\sigma_2]M,\sigma_1)$ if $s\sigma_1\sigma_2$ is a ground term and $\sigma_1 = \{x \mapsto r \mid x \notin fv(t)\}$, $\sigma_2 = \{x \mapsto r \mid x \in fv(t)\}$
5. $([t := s]M,\sigma) \rhd \perp\!\perp$ if $s\sigma$ is not a ground term or the terms t and $s\sigma$ do not unify
6. $([\top ?]M,\sigma) \rhd (M,\sigma)$
7. $([\perp ?]M,\sigma) \rhd \perp\!\perp$
8. $([A_1; A_2]M,\sigma) \rhd ([A_1][A_2]M,\sigma)$
9. $([A_1 :: A_2]M,\sigma) \rhd ([A_1]M,\sigma)$
10. $([A_1 :: A_2]M,\sigma) \rhd ([A_2]M,\sigma)$
11. $([A^*]M,\sigma) \rhd ([\top ? :: A; A^*]M,\sigma)$
12. $([\,]\,p,\sigma) \rhd \sigma$

Rules from (1) to (3) describe the equational definition. In (1), the unification of two constructor-based terms with identical constructor is converted into the equational problem $f(t_1,\ldots,t_n)\ \sigma = f(s_1,\ldots,s_n)\ \sigma$. If the problem has a solution, the solution is composed with the current state. In (2), the equation $f(t_1,\ldots, t_n) = g(s_1,\ldots, s_n)$ has no solution if f and g are different constructors or at most one of them have no definition as a function whose evaluation does not unify with the other. In (3), the equation $t = g(s_1,\ldots, s_n)$ can be solved if there is a functional definition of g and both the application $g(s_1\, \sigma,\ldots, s_n\, \sigma)$ and the term $t\sigma$ are equals to a term s. Rules (4) and (5) describe the assignment instruction. In (4), the assignment instruction succeeds if the instance of the term at the right-hand side under $\sigma_1\sigma_2$ is a ground term and all the variables occurring in the term at the left-hand side have become unbound from their values. In (5), the assignment fails either if $s\sigma$ is neither a ground term nor $s\sigma$ unifies with t. In (6), the guard true, named skip, always succeeds, whereas in (7) the guard false, named fail, always fails. In (8), the sequential execution of $A_1\, ;A_2$ corresponds to the execution of A_1 followed by the execution of A_2; otherwise $A_1\, ; A_2$ fails if either A_1 or A_2 fails. In (9) and (10), the non-deterministic execution $A_1 :: A_2$ of A_1 and A_2 corresponds to the selection of the unfailing execution of any of them. In the case that both succeeded, one of them is chosen, whereas if both fail, the instruction fails too. In (11), the iterative execution of A is obtained by non-deterministically choosing between skip and unwinding the iteration with the sequence $A; A^*$. Finally, in (12) the program modality terminates when no instructions remain to execute. In this case, if the program executed is partially correct with postcondition p, then $p\sigma$ holds.

6 Conclusions and Future Work.

In this paper the use of a restricted form of E-unification has been proposed for the imperative programming fragment of the DL Prolog language. E-unification extends unification of terms to include equational theories encoded as functions, gaining some control at the cost of limiting its expressive power. Nonetheless this reduced expressivity brings in more efficiency as required for most practical applications. E-unification is used to define logical variables by term unification and to redefine imperative variables by the assignment instruction.

Further research directions include: the proof of the soundness and completeness results of the imperative fragment, the design and implementation of DL Prolog through an extension to the WAM, and its comparison with other programming languages like standard Prolog in terms of their expressiveness and efficiency.

References

1. DeGroot, D., Lindstrom, G. (eds.): Logic Programming: Functions, Relations and Equations. Prentice-Hall, 1986.
2. Kahn, K.M.: Uniform : A Language Base upon Unification which unifies much of Lisp, Prolog and Act 1. In [1], 411-440.
3. Baader, F., Nipkow. T.: Term Rewriting and All That. Cambridge, 1998.
4. Antoy, S., Hanus, M.: Functional Logic Programming. Communications of the ACM, 53(4):74-85, 2010.
5. Sewell, P., Wojciechowski, P.T., Unyapoth, A.: Nomadic PICT: Programming Languages, Communication Infrastructure Overlays, and Semantics of Mobile Computation. ACM Transactions of Computer Languages, 32(4):12:1-12:63, 2010.

6. Van Roy, P.: Multiparadigm programming in Mozart/Oz. Second International Conference MOZ 2004, 2005.
7. Reynolds, J.C.: Theories of Programming Languages. Cambridge, 1998.
8. Olmedo-Aguirre, J.O., Morales-Luna, G.: A Dynamic-Logic-based Modal Prolog. In proceedings MICAI 2012. To appear.