Imperative Programming for Applications with built-in Intelligence

José Oscar Olmedo-Aguirre

Electrical Engineering, Cinvestav-IPN, Av. IPN 2508, Mexico City, Mexico 07360,
oolmedo@cinvestav.mx

Abstract. The convergence of diverse technologies has lead to increasingly complex applications that demand more built-in intelligence. Efficient programming languages like C lack of the right level of abstraction required for this purpose. The integration of the very high-level features of symbolic languages like Prolog combined with the efficiency of imperative languages like C would be welcomed with no doubt. DLProlog (Dynamic Logic Prolog) is an experimental language that integrates in a uniform programming model the elements required for the development of applications with built-in intelligence by coalescing the functional, logic and imperative programming paradigms. The key feature of DL-Prolog consists in the introduction of dynamic logic modalities into the head of the clauses. The modality encloses an efficient imperative program that uses high-level symbolic instructions like assignments based on E-unification. In this paper, the imperative fragment of DLProlog is 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. Logic Programming, Dynamic Logic, Knowledge Representation and Reasoning.

1 Introduction

The modern convergence of diverse technologies has lead to increasingly complex applications. This trend calls for an analogous convergence in programming concepts and models. 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, DLProlog [8], an extension to pure Prolog with dynamic logic modalities, 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 DLProlog stands more balanced 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 DLProlog.

This article is organized as follows. In section 2, a succinct review of the related work is presented. In section 3, a brief account of the programming style of the imperative fragment of DLProlog is shown. In section 4, a formal description in the structured operational semantics is given for the imperative fragment. Finally in section 5, 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 DLProlog 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 the 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 DLProlog 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 DLProlog unification prevents applying higher-order unification which is known to be 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 not 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 an appealing 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]. However, besides 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 for the most common applications would be this integration. Nonetheless, the concurrent programming approach has also leaded to one of the more successful pragmatic language designs as in Oz [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 Equational Reasoning within Imperative Programs

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]. Hereinafter, DLProlog variable names are written starting with the last alphabet letters u, v, w, x, y, z and lists of terms are written in the Prolog style. For a brief account of unification and E-unification, consider the following DLProlog program that involves lists of terms as in Prolog. 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 $[e, f, g, h]$.

$$
\begin{array}{l}
[\texttt{var}\, ys, zs:\; xs = [a,b,c,d]; ys := [e,f,g,h]; \\
\quad zs := rev(app(xs,ys)); us = zs; writeln(revapp : us); \\
\quad zs := app(rev(ys), rev(xs)); vs = zs; writeln(apprev : vs) \\
](us = vs).
\end{array}
$$

This DLProlog program consists of two parts: (i) the modal connective that encloses in brackets the *action* (i.e. a well-formed imperative program fragment): $\texttt{var}\, ys, zs : xs = [a, b, c, d], \cdots$, and (ii) the postcondition $us = vs$ that states the properties of the action that hold after its execution upon the values bound to the variables us and vs. The property is valid in a theory of lists with the usual operations $rev(xs)$ that reverses the order of the elements of list xs and $app(xs, ys)$ that appends the elements of list ys at the end of list xs. In the action, the logical variable xs is defined with list $[a, b, c, d]$, whereas the imperative variable ys is defined with list $[e, f, g, h]$. Logical variables are pure Prolog variables that can be defined by equality at most once, whereas imperative variables are DLProlog variables that can be arbitrarily redefined as many times as needed by means of the destructive assignment of imperative programming. Furthermore, logical variables can be introduced with no declaration as in Prolog, whereas imperative variables can be only introduced by declaration with the `var` binder. In any case, the occurrence of either a logical or imperative variable in an expression denotes the value bound to it if any.

In DLProlog, imperative instructions are sequentially executed as usual from left to right of the semicolon connective. In the action, 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 text, the scope of the binder `var` reaches its end and the imperative variables ys and zs cease their existence. Nonetheless, the values bound to the logical variables us and vs remain unaltered at the action postcondition $us = vs$. The partial correctness of the action ensures that the postcondition is always satisfied upon its termination whenever the input variables satisfy a suitable precondition. Hence there is no need to test the postcondition whenever the partial correctness of the action can be formally proved.

3.1 The Reverse Program

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

```
⎡ var $ys_1, ys_2$:                                                    ⎤
⎢   $(ys_1, ys_2) := ([\,], xs)$;                                      ⎥
⎢   while $ys_2 \neq [\,]$ do                                          ⎥
⎢     var $y$: $(y, ys_1, ys_2) := (\texttt{hd}(ys_2), [y \mid ys_1], \texttt{tl}(ys_2))$   ⎥  $rev(xs) = zs \Leftarrow list(xs)$
⎢   od;                                                                ⎥
⎣   $zs := ys_1$                                                       ⎦
```

where `hd` and `tl` are predefined functions that obtain the first and the rest of a list. The above program is in fact the partial correctness property of the program that computes the reverse function. Like Prolog, the reverse program is a Horn clause consisting of a single 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 is a dynamic logic assertion consisting of the reverse action enclosed in brackets followed by the postcondition $rev(xs) = zs$. A DLProlog action fails either if it does not terminate, or if upon its completion, it cannot satisfy its postcondition. As illustrated by the previous example, a DLProlog clause has the following structure that states the partial correctness relation of the action A:

$$[A]R \Leftarrow P$$

where P and R are the pre- and post-conditions of A. All the logical variables occurring in P are considered *input variables* in A, whereas the variables occuring only in R (and not in P) are considered *output variables*. Following the reverse program example, the only input variable is xs, whereas the only output variable is zs. In DLProlog, all input variables are treated as constants in action A, whereas the output variables can be defined at most once there. The input and output variables are visible to any instruction of A. DLProlog relies on the usual block programming construct to ensure both restricted lexical visibility for the instructions within the block and controlled allocation of local variables. The block ensures that new local variables become into existence when the execution enters into it and cease their existence when the execution leaves it, keeping them appart from all the input and output variables.

In the imperative programming model of DLProlog, imperative variables can be redefined, whereas all the logical variables cannot. However, both logical and imperative variables may freely occur in expressions, because the occurrence of a variable name denotes the value bound to it according to the current state. E-unification, assignment and resolution on a predicate goal may modify the state of the computation, whereas all the other programming constructs either evaluates expressions or alters the course of the action. In the block structure of DLProlog, all computations performed by the modality $[A]$ can only modify either the imperative variables or the output logical variables. Because output variables can be defined at most once, they are only used to assign the final values computed in A. Thus, the behavior of the imperative program fragment becomes encapsulated within the modal connective, being only visible its effects through the modification of the output variables occurring in R.

4 DLProlog Formal Description

Because most of the modern imperative languages are type-checked, DLProlog is formalized as a many-sorted first-order logic language to ensure a type checking discipline. Given a (Σ, Ξ)-structure, $\Sigma = \bigcup_{\alpha} \Sigma_{\alpha}$ is a family of *constructor* (constant) names and $\Xi = \bigcup_{\beta} \Xi_{\beta}$ is a family of *variable* names, each partitioned by the basic types (sorts) `bool` and `nat`, among possibly others. In the following presentation, the name `var` can be replaced by any of the basic types. The set $T(\Sigma, \Xi)$ of *terms with variables* is the minimal set of phrases that is closed under composition of a constructor with a (possibly empty) sequence of terms. The set $T(\Sigma) = T(\Sigma, \emptyset)$ of *ground terms* consists of terms with no variables. *Type inference* is embedded in the grammar rules of the language to ensure that clauses and programs are well-formed. In particular, the grammar rule shown below describes the syntactic structure of a well-formed term T_β of type β:

Type inference $T_\beta ::= x_\beta \mid c_\beta \mid c_{\beta_1 \cdots \beta_n \to \beta}(T_{\beta_1}, \ldots, T_{\beta_n}) \mid f_{\beta_1 \cdots \beta_n \to \beta}(T_{\beta_1}, \ldots, T_{\beta_n})$

In that follows, all type annotations are ommited for the sake of brevity. The set $P(\Sigma, \Xi)$ of *atomic predicates with variables* is the minimal set closed under composition of predicate symbols with (possibly empty) sequences of terms. The set $P(\Sigma) = P(\Sigma, \emptyset)$ of *ground atoms* consists of all atoms with no variables. A *literal* is an atom or a negated atom. A *clause* is a disjunction of literals. The set of clauses with variables is denoted $C(\Sigma, \Xi)$. Clauses are usually written in implication form $P \Leftarrow Q$, where P, called the *consequent,* is a disjunction of atoms and Q, called the *antecedent*, is a conjunction of atoms. A *unit clause* contains only one literal. A *positive clause* contains no negated atoms, whereas a *negative clause* contains no positive atoms. A *Horn clause* contains at most a positive atom. A *goal* consists only of negative atoms that can be represented by an implication with false as consequent. The logical variables occurring in a clause are universally quantified. The *set of goals with variables* is denoted $G(\Sigma, \Xi)$. In that follows, in any context where the term list $T_1, \ldots, T_n$ may occur, the restriction $n \geq 0$ is always assumed for n.

Terms	$T ::= x \mid c \mid c(T_1, \ldots, T_n) \mid f(T_1, \ldots, T_n) \mid (T_1, \ldots, T_n)$
Lists	$L ::= [\,] \mid [\ T_1, T_2, \ldots, T_n \mid L\ ]$
Predicates	$P ::= \texttt{false} \mid \texttt{true} \mid T_1 = T_2 \mid p(T_1, \ldots, T_n)$
Goals	$G ::= P \mid G_1 \wedge G_2$
Clauses	$B ::= P \mid P \Leftarrow G \mid \forall x.B$
Actions	$A ::= \texttt{skip} \mid \texttt{fail} \mid T_1 := T_2 \mid G\texttt{?} \mid$
	$A\texttt{*} \mid A_1\,\texttt{;}\,A_2 \mid A_1 \cup A_2 \mid \texttt{var}\ x_1, \ldots, x_n :\ A \mid (A)$
Modal actions	$M ::= P \mid\ [A]\ M \mid \langle A \rangle\ M$
Modal clauses	$F ::= M \mid G \Rightarrow M \mid \forall x.F$

The set $A(\Sigma, \Xi)$ *of actions with variables* is the minimal set of actions that is closed under composition according to the following action connectives. A basic action is either the null action (`skip`), the failure (`fail`), the assignment of terms to variables by using the binary operator (:=), or the postfix unary operator for condition testing (`?`). The action connectives, in descending order of precedence, are the postfix unary operator for iteration (`*`), the infix binary operator for sequential composition (`;`), and the infix binary operator for non-deterministic choice ($\cup$). The declaration of the imperative variables $x_1, \ldots, x_n$ in A by the binder `var` introduces variables in the current block with a scope that extends as far as possible to the right. The precedence of the action connectives and the extension of the scope can be altered by grouping (A). The mixfix modal connectives of modal necessity ($[\,]$) and possibility ($\langle\,\rangle$) compose actions along with their postconditions. The following equalities between actions provide an interpretation of the usual imperative programming constructs:

$$\begin{aligned} \texttt{skip} \equiv \texttt{true?}\ \ ,\ \ \texttt{fail} &\equiv \texttt{false?} \\ \texttt{if}\ F\ \texttt{then}\ A\ \texttt{fi} &\equiv F\texttt{?;}A \\ \texttt{if}\ F\ \texttt{then}\ A_1\ \texttt{else}\ A_0\ \texttt{fi} &\equiv (F\texttt{?;}A_1) \cup (\neg F\texttt{?;}A_0) \\ \texttt{while}\ F\ \texttt{do}\ A\ \texttt{od} &\equiv (F\texttt{?;}A)\texttt{*;}\neg F\texttt{?} \end{aligned}$$

The unification $T_1 = T_2$? between terms T_1 and T_2 is a test for equality and it can be written with no test operator `?` if followed by a sequential composition connective. The assignment notation $T_1 := T_2$ generalizes the single assigment

$x := T$ and the multiple assignment $x_1, \ldots, x_n := T_1, \ldots, T_n$ to arbitrary equally structured terms T_1 and T_2. Their precise meaning will be given later on by means of the SOS semantics of DLProlog.

In a signature (Σ, Ξ) with variables, a *substitution* is a partial function $\sigma : \mathit{\Xi} \to T(\Sigma, \Xi)$, where $\sigma(x) \neq x$ for any variable $x \in \mathit{\Xi}$. $\{\}$ denotes the empty substitution. A *ground substitution* is a substitution $\sigma : \mathit{\Xi} \to T(\mathit{\Sigma})$ valued on ground terms. For any variable $x \in \mathit{\Xi}$ and any substitution σ, let $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$ and $x\sigma = x$ otherwise. For any term $t \in T(\Sigma, \Xi)$, let $t\sigma$ be the term obtained by substituting any variable x appearing in T by $x\sigma$:

$$\begin{aligned}
x\{\} &= x \\
x\,\sigma &= \begin{cases} x & \text{if } x \notin \mathrm{dom}(\sigma) \\ \sigma(x) & \text{if } x \in \mathrm{dom}(\sigma) \end{cases} \\
c\,\sigma &= c \\
c(T_1, \ldots, T_n)\,\sigma &= c(T_1\,\sigma, \ldots, T_n\,\sigma) \\
[A]\,p\,\{\} &= [A]\,p \\
[A]\,p\,\sigma &= [\sigma_{:=}\,; A]\,p
\end{aligned}$$

where notation $[\sigma_{:=}]$ stands for the multiple assignment $x_1, \ldots, x_n := T_1, \ldots, T_n$ obtained from the substitution $\sigma = \{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$. Thus the substitution for a modal action A is defined as the initial value that the variables take before the action starts its execution. The *composition* of two substitutions $\sigma_0, \sigma_1 \in \mathit{\Xi} \to T(\Sigma, \Xi)$, written $\sigma_0 \cdot \sigma_1$, is defined as

$$\sigma_0 \cdot \sigma_1 : x \mapsto \begin{cases} (x\sigma_0)\sigma_1 & \text{if } x\sigma_1 \notin \mathrm{dom}(\sigma_1) \\ x\sigma_1 & \text{if } x \in \mathrm{dom}(\sigma_1) - \mathrm{dom}(\sigma_0) \\ \mathit{failure} & \text{otherwise} \end{cases}$$

Besides the natural extension to terms $T(\Sigma, \Xi) \to T(\Sigma, \Xi)$, substitutions are also extended to predicates, goals, and both backward and forward rules.

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:

$$\begin{aligned}
(\{t = t\} \cup P, S) &\to (P, S) \\
(\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup P, S) &\to (\{t_1 = s_1, \ldots, t_n = s_n\} \cup P, S) \\
(\{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \cup P, S) &\to (\{\}, \{\bot\}) \\
(\{x = t\} \cup P, S) &\to (\{\}, \{\bot\}) \ \text{ if } \ x \in fv(t) \text{ and } x \neq t \\
(\{x = t\} \cup P, S) &\to (P\,\{x \mapsto t\}\,, S\,\{x \mapsto t\} \cup \{x = t\}) \\
&\quad \text{if } \ t \notin \mathit{\Xi}, x \notin fv(t)
\end{aligned}$$

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

The semantic description of DLProlog 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* 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* corresponds to the state reached when the program either terminates or fails. The final description of a program that terminates successfully is a state $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, 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 $\perp$. A *transition* is a relation $\triangleright$ over pairs of program configurations. An *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 de-scriptions ended by a final description. For the successful program execution, we have, for some $n \geq 0$:

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

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:

$$(M_0, \sigma_0), (M_1, \sigma_1), \ldots, (M_i, \sigma_i), (M_{i+1}, \sigma_{i+1}), \ldots, (M_n, \sigma_n), \perp$$

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:

$$\frac{(\{f(t_1, \ldots, t_n)\sigma = f(s_1, \ldots, s_n)\sigma\}, \{\}) \to (\{\}, \{x_1 = r_1, \ldots, x_m = r_m\})}{([f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)]\, M, \sigma) \triangleright (M, \sigma\, \{x_1 \mapsto r_1, \ldots, x_m \mapsto r_m\})} \tag{1}$$

$$([f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)]\, M, \sigma) \triangleright \perp \tag{2}$$

$$\frac{(\{s = t\sigma\}, \{\}) \to (\{\}, \{x_1 = r_1, \ldots, x_n = r_n\})}{([f(s_1, \ldots, s_m) = t]\, M, \sigma) \triangleright (M, \sigma\, \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\})} \tag{3}$$

$$([t := s]\, M, \sigma_1\sigma_2) \triangleright ([t = s\sigma_1\sigma_2]\, M, \sigma_1) \tag{4}$$

$$([t := s]\, M, \sigma) \triangleright \perp \tag{5}$$

$$([\top?]\, M, \sigma) \rhd (M, \sigma) \tag{6}$$

$$([\bot?]\, M, \sigma) \rhd \bot \tag{7}$$

$$([A_1; A_2]\, M, \sigma) \rhd ([A_1]\, [A_2]\, M, \sigma) \tag{8}$$

$$([A_1 \cup A_2\,]\, M, \sigma) \rhd ([A_1]\, M, \sigma) \tag{9}$$

$$([A_1 \cup A_2\,]\, M, \sigma) \rhd ([A_2]\, M, \sigma) \tag{10}$$

$$([A*]\ M, \sigma) \rhd ([\top?\ \cup A; A*]\, M, \sigma) \tag{11}$$

$$([\,]\, P, \sigma) \rhd \sigma \tag{12}$$

Rules from (1) to (3) describe unification and E-unification. In (1), the unification of two constructor-based terms with identical constructor is converted into the equational problem $f(t_1, \ldots, t_n)\sigma = f(s_1, \ldots, s_n)\sigma$ that is the instance of the equation under σ. Whenever the unification 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 either f and g are different constructors or there are not defined functions for at least one of them. 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 equal to some ground term s. Rules (4) and (5) describe the assignment instruction. In (4), if there is a function defined for f and if $s\sigma_1\sigma_2$ is a ground term $\sigma_1 = \{x \mapsto r | x \notin fv(t)\}$ and $\sigma_2 = \{x \mapsto r | x \in fv(t)\}$, 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 been unbound from their previous values if any. In (5), the assignment fails either if $s\sigma$ is neither a ground term nor $s\sigma$ unifies with t. In (6), the action `skip` corresponds to the guard `true`?, always terminates successfully, whereas in (7) the action `fail` corresponds to the guard `false`? does not terminate and if it does, it reaches an state where no condition holds. In (8), the sequential execution of $A_1; A_2$ corresponds to the execution of A_1 followed after its completion 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 $A_1 \cup A_2$ 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 arbitrarily chosen, whereas if both fail, the instruction fails too. In (11), the repetitive execution of A is obtained by non-deterministically choosing between `skip` and unwinding the iteration by means of 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 w.r.t. postcondition P, then $P\sigma$ holds.

5 Conclusions and Further Work

In this paper we have described some of the programming features of DLProlog, a dynamic logic modal extension to the pure Prolog programming language and model, in order to preserve its declarative nature without compromising its expressiveness. An experimental compiler and interpreter has been already written in LPA Prolog [9] to provide some evidences about the improvements on the readability and efficiency of DLProlog over both standard and pure Prolog. There is a number of appealing research directions from the proposed DL modal extension:

1. Extending the unification procedure to other constraint satisfaction methods including other domains like finite domain constraints seems to be possible. However, for such an extension, don't known non-determinism must be included in the semantics of actions.
2. Extending the imperative language and programming model with mechanisms for encapsulation and inheritance to introduce the fundamental notions of objects and classes.
3. Using popular imperative languages like Java or C# instead of the algorithmic language used here that is reminiscent of the Algol68 programming language.
4. Designing and implementing an extension of the standard WAM to include low level instructions like destructive assignment and conditional jumping. A substantial contribution may arise from this research direction.

Due to its inherited symbolic processing capabilities, DLProlog is a very high-level programming language thay brings the best of the successful imperative and functional-logic paradigms for programmers that need to develop modern applications with built-in intelligence.

6 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, CPS IEEE Computer Society. pp. 3-9. ISBN: 978-0-7695-4904-0.
9. LPA Prolog, Logic Programming Associates, url: http://www.lpa.co.uk/