Operational Approach to the Modified Reasoning, Based on the Concept of Repeated Proving and Logical Actors

Alexei A. Morozov

Institute of Radio Engineering and Electronics RAS
Mokhovaya 11, Moscow, Russia, 125009
morozov@cplire.ru
http://www.cplire.ru/Lab144/

Abstract. The message of this paper is the following: there is one more basic principle of operational semantics of logic programming (besides backtracking, recursion, etc.) that gives a solution of challenging problem of combining strict declarative semantics of logic languages with the dynamic behavior (that includes destructive assignment operations and interaction with dynamic environment). We have developed this principle, named repeated proving, in the Actor Prolog logic language. In this paper the repeated proving principle is explained with the help of an operational semantics (abstract machine) for sequential logic programs enhanced with logical actors. The problems of soundness and completeness of the control strategy are considered.

Introduction

We address the problem of ensuring strict declarative semantics of logic languages operating in dynamic environment [1,2,3,4]. Our approach reminds of so-called perturbation model of constraint-based languages. In the perturbation model, unlike the standard (refinement) one, at the beginning of execution cycle variables have specific associated values satisfying the constraints. The value of one or more variables is perturbed by some outside influence, such as an edit request from the user, and the task of the prover is to adjust the values of the variables in such a way as to satisfy the constraints again [5,6].

The problem is closely related to the problem of ensuring the declarative semantics of the destructive assignment operation in logic languages. One can consider the updates in the outer world as a kind of destructive assignment that violates the soundness of the logic program. In this article, this problem is solved using the principle of repeated proving of sub-goals.

In section 1, the ideas of repeated proving and logical actors are set forth. In section 2, a special notation is introduced along with the architecture of an abstract machine implementing a sequential control strategy of logic programs enhanced with logical actors. In section 3, transition diagrams of the abstract machine are defined. In section 4, the problems of soundness and completeness of the operational semantics are discussed.

1 The Idea of Repeated Proving and Logical Actors

Let us consider a logic program written in pure Prolog that has a classical model-theoretic semantics. The idea of repeated proving consists in dividing the AND-tree of the logic program into separate branches (sub-goals to be proved) called logical actors (α_1,...,α_n on the Fig. 1) that should have the following operational properties:

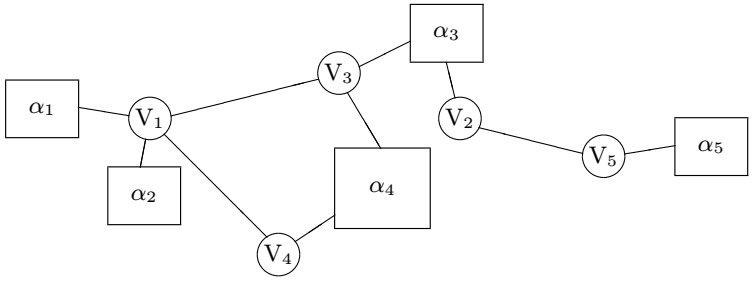

Fig. 1. The idea of repeated proving of sub-goals.

1. Common variables (V_1,...,V_m) are the single channel of data exchange between the actors.
2. Proving of separate actors can be fulfilled independently in arbitrary order.
3. One can cancel the results of proving of some actors *without* logic program backtracking while keeping all other sub-goals of the program. After canceling results of proving of an actor, its proving is to be repeated.

Thus, one can implement a modification of reasoning. The results and consecution of reasoning itself can be partially modified in the process and after the logical inference. This makes possible to eliminate contradictions between the results of logical reasoning and new information income from outer world.

The best example of application of the idea is implementation of long-lived Web agents. Let us imagine a Web agent written in logic language. The purpose of the agent is to make a logical inference on the basis of several remote data sources and to check some assertions about the remote resources. Let us imagine also that the agent is long-lived, i.e., it operates during a period of time that is *longer* than the period of information update. Thus the agent should react on any modification of remote resources and inform the user about the current state of the assertions to be checked. The problem is that one cannot repeat execution of the logic program from the beginning with any change in the outer world — the repeat of the whole process of data collection performed during the long period of time is inefficient and, in some cases, technically impossible. Therefore one must change some branches of logic inference that depend on the modified data and keep all other branches unchanged. This is the case of modification of reasoning and the challenge is to provide soundness and (if possible) completeness of logical reasoning under the modification.

Another area that is recognized as a prospective application of the perturbation model of constrain-based languages is graphic user interface management [6]. We have successfully applied the logical actors approach for both the logical programming of Web agents [7,8] and visual user interface management [9]. An additional issue of our research is development of logic object-oriented model of asynchronous concurrent computations based on the logical actors approach [10].

In the following sections a conservative extension of standard control strategy of (sequential) Prolog is developed that implements the repeated proving of logical actors.

2 The Architecture of Abstract Machine

Let us consider an abstract machine that implements a sequential control strategy for logic programs enhanced with logical actors. The input language of this machine is the Horn subset of first order logic formulas enhanced with special means implementing logical actors.

The abstract machine implements the following general principles:

1. The standard control strategy (depth-first left-to-right search) is a part of the control strategy implemented by the abstract machine.
2. The AND-tree of logic program is to be divided into separate logical actors, i.e., any pending sub-goal of the program is a logical actor or a part of a logical actor.
3. Any logical actor obtains its own (local) substantiation (local values of common variables).
4. The results of proving of logical actor can be cancelled.
5. The logical actor can be proved once again after the canceling of results of its previous proving.
6. The states of logical actors are restored during the backtracking.

Thus, the abstract machine implements the standard control strategy exactly if there is only one logical actor in the program (i.e., all the branches of the AND-tree belong to the same actor).

Each logical actor A of the program has its own (local) values of variables. Actor A unifies its values with the values that belong to other actors in the following cases only:

1. The local values are compared in the course of successful termination of proving of actor A.
2. The local values are compared when actor A executes the *':='* built-in predicate (this predicate will be considered later).

During the comparison of values that belong to different actors, abstract machine can cancel results of proving of some actors to provide consistency between remaining actors of the program (to provide existence of the most general unifier for all the values of all the actors of the program that remain uncancelled). After that the abstract machine tries to prove the cancelled actors once again.

Let us name the operation of canceling of results of proving of the actor as *neutralization of actor*.

Thus, the proving of actor A includes the following main stages (see Fig. 2).

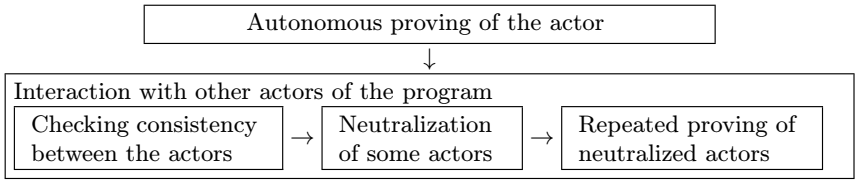

Fig. 2. The stages of execution of logical actor.

There are three possible states of the actor:

1. Let us name an actor *active* if the proving of this actor is performing at this moment and is not ended yet.
2. The actor that was successfully proved (and was not neutralized yet) is named *proven*.
3. The actor is named *neutral* if the proving of this actor was cancelled and the repeated proving of it was not started yet.

Neutralization of active actors is prohibited (see the formal rules of selecting actors for neutralization in section 3). Thus, sometimes the contradictions between the actors of the program cannot be eliminated with the help of actor neutralization. In this case standard backtracking occurs in the program, that returns actor A to the stage of autonomous proving.

In the case if the abstract machine successfully eliminates contradictions between the actors with the help of neutralization of some set NA of actors, repeated proving of all the neutral actors occurs. If proving of all neutral actors terminates with success (or set NA is empty), the proving of actor A terminates with success. In another case backtracking occurs in the logic program, that returns actor A to the stage of autonomous proving. Thus, a failure of the repeated proving of any actor of the NA set will backtrack the program.

Let us introduce some special notions to define the control strategy formally:

– *The state of abstract machine* is a set of actors:

$$\Gamma = \{A_1, A_2, \ldots, A_n\},$$

where A_i, $i = 1 \ldots n$, are the actors of the program.

– *Actor* A_i is a branch of AND-tree created as a result of execution of so-called *actor call* of a predicate $@m(t_1, \ldots, t_k)$:

$$A_i = \langle \alpha, m(t_1, \ldots, t_k), R \rangle,$$

where α is an (unique) name of actor; $m(t_1, \ldots, t_k)$ is an atomic formula that corresponds to given actor; R is a list named *the results of proving* of the actor.

- *The result of proving* of an actor is information obtained during the proving of the actor: instantiations of variables, backtrack points, etc.:

$$E = \langle \beta, F \rangle ,$$

where β is the name of actor that has invoked the proving under consideration; F is a stack of so-called *failure continuations* that is used for implementation of backtracking.

- *The failure continuation* is a stack containing sub-goals to be proved during investigation of one branch of OR-tree:

$$C = \langle G, \sigma, N, B \rangle ,$$

where G is *a list of sub-goals*; σ is an instantiation of variables used during investigation of the branch of the OR-tree under consideration; N is a list of actor names that were neutralized during investigation of given branch of the OR-tree; B is a list of actor names that were created during investigation of this branch.

- *The Subgoal* can be a usual predicate call $m(t_1, \ldots, t_k)$, an *actor* predicate call $@m(t_1, \ldots, t_k)$, compositions of sub-goals S_1 *and* S_2, S_1 *or* S_2, etc.

A special notation (@-language) necessary for definition of abstract machine states is given in tables 1, 2.

The semantics of formulas of kind $\Gamma.\alpha\,\{GL = S : G, Subst = \sigma\}$ is the following: there is an actor α in the Γ state of abstract machine, that has the following properties:

1. The GL cell situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor has value $S : G$ (a list).
2. The $Subst$ cell situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor has value σ.

In a similar manner, a formula of kind $\Gamma.\alpha = \langle \alpha, M, R \rangle$ has the following semantics: there is an actor α in state Γ of abstract machine. The value of this actor is equal to $\langle \alpha, M, R \rangle$. One can use given formulas in the following sense: "The Γ state, such that there is an actor α that has the following properties...".

We will use also formulas of the following kind in the transition diagrams: $\Gamma' = \Gamma : \alpha\,\{GL := G\}$. The semantics of these formulas is "The Γ' state of abstract machine differs from the Γ state in that a new value G was assigned to the GL cell that is situated on the top of the stack of failure continuations that is situated on the top of the stack of results of proving of the α actor."

Table 1. The table of basic symbols of the @-language.

Notion	**Symbol**	**Definition**	**Typical elements**
Constant	$Const$		a, b, c
Variable	Var		X, Y, Z
Functor	Fun		f
Term	$Term$	$Const$; Var; $f(t_1, \ldots, t_k),\ k \geq 1$	t, v, u
Atomic formula	$Atom$	$m(t_1, \ldots, t_k),\ k \geq 0$	M
Name of actor	$Name$	$\alpha, \beta, \gamma, \ldots; \tau; \xi$, where τ and ξ are special names	
Sub-goal	$Subgoal$	$true$; $fail$; M; $@M$; S_1 *and* S_2; S_1 *or* S_2; $del([\alpha_1, \ldots, \alpha_n])$; $back([\alpha_1, \ldots, \alpha_n])$; $wait(\gamma)$; $redo(\gamma)$; $neutralize(\{\alpha_1, \ldots, \alpha_n\})$; $restart(\{\alpha_1, \ldots, \alpha_n\})$	S
Procedure	$Procedure$	$M :- S$	P
Definition of procedures	$Procedures$	Function $Atom \rightarrow Subgoal$	D

Table 2. Definition of the @-language.

Notion	**Symbol**	**Definition**	**Typical elements**
State of abstract machine	$State$	$\{A_1, A_2, \ldots, A_n\},\ n \geq 1$	Γ
Actor	$Actor$	$\langle \alpha, M, R \rangle$	A
List of results of proving	RL	nil; $E : R$, is a list with head E and rest R.	R
Results of one proving	$Result$	$\langle \beta, F \rangle$; $neutral$ where $neutral$ is a special symbol	E
Stack of failure continuations	FL	nil; $C : F$	F
Failure continuation	$Cont$	$\langle G, \sigma, N, B \rangle$	C
List of sub-goals (named also success continuation)	GL	nil; $success$; $failure$; $S : G$	G
Substitution	$Subst$	$\sigma, \theta, \ldots; \varepsilon$ (ε is the empty substitution)	
List of names of neutral actors	$Neutr$	$[\alpha_1, \ldots, \alpha_n]$	N
List of names of created actors	$Built$	$[\alpha_1, \ldots, \alpha_n]$	B

A logic program is defined as a set D of procedures[1] (see designations of the @-language in table 1) and an initial state of the program:

$$\Gamma^0(\tau) = \left\langle \tau, m(t_1, \ldots, t_k), \left\langle \xi, \langle m(t_1, \ldots, t_k) : nil, \varepsilon, [\,], [\,]\rangle : nil \right\rangle : nil \right\rangle,$$

where τ is the name of an actor (the target actor hereafter) that is active in the Γ state, and ξ is dummy name of an actor situated in outer world (the external actor) that has invoked the program under consideration. All the actors except for the τ actor are *proven*[2] in the Γ^0 state of the abstract machine:

$$\forall \alpha : \Gamma^0.\alpha\,\{Name \neq \tau\} : is\_proven(\Gamma^0, \alpha)$$

The abstract machine can reach one of two final states:

1. The *success state*: $\Gamma^{SUCCESS}.\tau\,\{Cont = \langle success, \sigma, N, B\rangle\}$, where τ is the target actor introduced in the Γ^0 initial state.
2. The *failure state*: $\Gamma^{FAILURE}.\tau\,\{FL = \langle failure, \varepsilon, [\,], [\,]\rangle : nil\}$.

Note, that the *success* and the *failure* states are alternative in accordance with given definitions. Deadlocks never occur in the abstract machine.

3 Transition System

The transition system of abstract machine is defined with the help of set of transition schemas and set Λ of labels (let us denote the typical label by l).

Let us consider the main stages of the proving of logical actor (Fig. 2).

3.1 Autonomous Proving of Actor

Execution of logic program is performed in accordance with the standard control strategy (depth-first left-to-right search) on this stage of proving of the actor. This strategy is implemented with the help of the transition schemas: $True$, Rec, Loc_1, Seq, and Alt. Some auxiliary schemas implement creation, deletion, and modification of logical actors during the proving.

$True$ — elimination of the *true* sub-goal during the execution of actor α.

$$\Gamma.\alpha\,\{GL = true : G\} \xrightarrow{\langle True, \alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

The semantics of this transition schema is the following one: "If current state Γ of abstract machine is such that an actor α exists and current list of sub-goals of this actor $GL = true : G$, then state Γ can be transformed into new one. In new state of abstract machine current list of sub-goals of actor α is modified:

[1] Let us do not use different procedures with the same functor (name and arity) of heading M to simplify the presentation.

[2] The is_proven predicate is defined in section 3.2.

$GL := G$. All other attributes of actor α and all other actors of the abstract machine will not changed during the transformation."

Rec — a call of predicate m during the execution of actor α. The $rename : P \to P'$ function implements renaming of variables of given procedure in the standard manner. The $mgu : (M_1, M_2) \to \sigma$ function computes the most general unifier of terms M_1, M_2 (iff the unifier exists).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = m(t_1,\ldots,t_k) : G, Subst = \sigma\}\\ \exists\, P \in D :\\ \quad rename(P) = (m(u'_1,\ldots,u'_k) :\!-\, S') \;\wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, m(u'_1,\ldots,u'_k))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := S' : G, Subst := \sigma\theta\}\\ \Gamma \xrightarrow{\langle Rec,\alpha\rangle} \Gamma'\end{array}}$$

where $\langle Rec, \alpha\rangle$ is the label of transition scheme under consideration. The statements over the line determine the conditions when the *Rec* schema can be performed. The statements under the line explain what is the difference between old state Γ and new state Γ' that can be obtained with the help of the *Rec* transition schema.

Loc_1 — backtracking of given actor α. The '$-$' function designates the difference between lists: $L - L' = L''$, if $L'' = [\alpha_1,\ldots,\alpha_n]$ and $L = [\alpha_1,\ldots,\alpha_n|L']$. The '$+$' function designates concatenation of lists.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : (\langle G', \sigma', N', B'\rangle : F')\}\,,\\ S = fail \vee\\ (S = m(t_1,\ldots,t_k) \;\wedge\; \neg\exists\, P \in D :\\ \quad rename(P) = (M' :\!-\, S') \;\wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N'' + B'') : (del(B'') : G')\,, \sigma', N', B'\rangle : F'\}\,,\\ N'' = N - N',\; B'' = B - B'\\ \Gamma \xrightarrow{\langle Loc_1,\alpha\rangle} \Gamma'\end{array}}$$

Loc_2 — recognition of necessity to transmit backtracking from actor α to the actor that has invoked current proving of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle S : G, \sigma, N, B\rangle : \underline{nil}\}\,,\\ S = fail \vee\\ (S = m(t_1,\ldots,t_k) \;\wedge\; \neg\exists\, P \in D :\\ \quad rename(P) = (M' :\!-\, S') \;\wedge\\ \quad \exists\,\theta = mgu(m(t_1,\ldots,t_k)\sigma, M'))\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{FL := \langle back(N + B) : (del(B) : failure)\,, \varepsilon, [\,], [\,]\rangle : nil\}\\ \Gamma \xrightarrow{\langle Loc_2,\alpha\rangle} \Gamma'\end{array}}$$

Glo — transmission of backtracking from actor α to actor β.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Result = \langle\beta, \langle failure, \varepsilon, [\,], [\,]\rangle : nil\rangle\}\\ \Gamma.\beta\,\{Subgoal = wait(\alpha)\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := fail : nil\}\\ \Gamma \xrightarrow{\langle Glo,\beta,\alpha\rangle} \Gamma'\end{array}}$$

$Back_0$ — termination of process of recovering the states of actors during backtracking of the program.

$$\Gamma.\alpha\,\{GL = back([\,]) : G\} \xrightarrow{\langle Back_0,\alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

$Back_1$ — recovery of the *active* or the *proven* state of actor γ during backtracking of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\}\\ \Gamma.\gamma\,\{RL = \langle\beta, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : F_\gamma\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(N_\gamma + B_\gamma + BList) : (del(B_\gamma) : G_\alpha)\}\,,\\ \qquad\quad \gamma\,\{RL := R_\gamma\}\end{array}}$$
$$\Gamma \xrightarrow{\langle Back_1,\alpha,\gamma\rangle} \Gamma'$$

$Back_2$ — recovery of the *neutral* state of actor γ during backtracking of α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = back([\gamma|BList]) : G_\alpha\}\\ \Gamma.\gamma\,\{RL = neutral : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := back(BList) : G_\alpha\}\,,\\ \qquad\quad \gamma\,\{RL := R_\gamma\}\end{array}}$$
$$\Gamma \xrightarrow{\langle Back_2,\alpha,\gamma\rangle} \Gamma'$$

Del_0 — termination of deletion of actors during backtracking of actor α.

$$\Gamma.\alpha\,\{GL = del([\,]) : G\} \xrightarrow{\langle Del_0,\alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

Del_1 — deletion of actor γ during backtracking of actor α. The '/' function designates deletion of actor: $\Gamma_1/\gamma = \Gamma_2$, such that $\{\langle\gamma, M_\gamma, R_\gamma\rangle\} \cup \Gamma_2 = \Gamma_1$, $\Gamma_1 \neq \Gamma_2$.

$$\frac{\Gamma.\alpha\,\{Subgoal = del([\gamma|DList])\}}{\Gamma' = (\Gamma : \alpha\,\{Subgoal := del(DList)\})\,/\gamma}$$
$$\Gamma \xrightarrow{\langle Del_1,\alpha,\gamma\rangle} \Gamma'$$

Seq — execution of conjunction of sub-goals of actor α.

$$\Gamma.\alpha\,\{GL = (S_1\ and\ S_2) : G\} \xrightarrow{\langle Seq,\alpha\rangle} \Gamma.\alpha\,\{GL := S_1 : (S_2 : G)\}$$

Alt — execution of disjunction of sub-goals of actor α.

$$\Gamma.\alpha\,\{FL = \langle(S_1\ or\ S_2) : G, \sigma, N, B\rangle : F\} \xrightarrow{\langle Alt,\alpha\rangle} \Gamma.\alpha\,\{FL := \langle S_1 : G, \sigma, N, B\rangle : (\langle S_2 : G, \sigma, N, B\rangle : F)\}$$

New_1 — execution of actor predicate call $@m$ during execution of actor α.

The *code* auxiliary function is used for preparation of arguments of actor predicate call. This function (see Fig. 3) provides transfer of maximal quantity of information about the values of the arguments of predicate into the γ actor

to be created. The *code* function transfers the values of the instantiated variables and copies the variables that are unbound. The *copy* auxiliary function copies the values of variables. The *new_variable* function creates new variables. The *is_variable* function checks if the argument is an (unbound) variable. The $not\_exists(\Gamma,\gamma)$ expression means $\langle\gamma, M_\gamma, F_\gamma\rangle \notin \Gamma$.

$$
\begin{array}{l}
\Gamma.\alpha\,\{FL = \langle @m(t_1,\ldots,t_k) : G,\sigma,N,B\rangle : F\} \\
not\_exists(\Gamma,\gamma) \\
\exists\, P \in D : \\
\qquad rename(P) = (m(u'_1,\ldots,u'_k) :- S') \ \wedge \\
\qquad ([v_1,\ldots,v_k]\,,\sigma') := code([t_1,\ldots,t_k]\,,\sigma) \wedge \\
\qquad \exists\,\theta = mgu(m(v_1,\ldots,v_k), m(u'_1,\ldots,u'_k)) \\
\hline
\Gamma' = \left(\Gamma : \alpha \left\{ \begin{array}{l} FL := \langle wait(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle \\ : (\langle redo(\gamma) : G, \underline{\sigma'}, N, [\gamma|B]\rangle : F) \end{array} \right\}\right) \cup \\
\qquad \{\langle\gamma, m(v_1,\ldots,v_k), \langle\alpha, \langle S' : nil, \underline{\theta}, [\,], [\,]\rangle : nil\rangle : nil\rangle\} \\
\Gamma \xrightarrow{\langle New_1,\alpha,\gamma\rangle} \Gamma'
\end{array}
$$

$code : [\{t_i\},\sigma] \to [\{t'_i\},\sigma']\,, i = 1\ldots n$	$copy : [t,\sigma] \to [t',\sigma']$
$\sigma' := \sigma;$	$if\ \ t\sigma = f(\{u_j\}), j = 1\ldots k$
$do\ \ i = 1\ldots n$	$\quad \sigma' := \sigma;$
$\quad if\ \ t_i = f(\{u_j\}), j = 1\ldots k$	$\quad do\ \ j = 1\ldots k$
$\quad\quad [\{v_j\},\sigma'] := code(\{u_j\},\sigma);$	$\quad\quad if\ \ is\_variable(u_j)$
$\quad\quad t'_i := f(\{v_j\});\ \ \sigma := \sigma'$	$\quad\quad\quad u'_j := new\_variable();$
$\quad elsif\ \ is\_variable(t_i)$	$\quad\quad\quad \sigma' := \sigma \cup \{u'_j = u_j\}$
$\quad\quad if\ \ is\_variable(t_i\sigma)\ \ t'_i := t_i$	$\quad\quad else\ \ [u'_j,\sigma'] := copy(u_j,\sigma)$
$\quad\quad else\ \ [t'_i,\sigma'] := copy(t_i,\sigma);$	$\quad\quad fi;$
$\quad\quad\quad \sigma := \sigma'$	$\quad\quad \sigma := \sigma'$
$\quad\quad fi$	$\quad od;$
$\quad else\ \ t'_i := t_i$	$\quad t' := f(\{u_j\}), j = 1\ldots k$
$\quad fi$	$else\ \ t' := t\sigma;\ \ \sigma' := \sigma$
od	fi

Fig. 3. Definitions of coding and copying functions.

New_2 — recognition of that an actor predicate $@m$ call cannot be performed during the execution of actor α.

$$
\begin{array}{l}
\Gamma.\alpha\,\{Subgoal = @m(t_1,\ldots,t_k), Subst = \sigma\} \\
\neg\exists\, P \in D : \\
\qquad rename(P) = (m(u'_1,\ldots,u'_k) :- S') \ \wedge \\
\qquad ([v_1,\ldots,v_k]\,,\sigma') := code([t_1,\ldots,t_k]\,,\sigma) \wedge \\
\qquad \exists\,\theta = mgu(m(v_1,\ldots,v_k), m(u'_1,\ldots,u'_k)) \\
\hline
\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\
\Gamma \xrightarrow{\langle New_2,\alpha\rangle} \Gamma'
\end{array}
$$

$Redo_1$ — backtracking of the γ actor during backtracking of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha\} \\ \Gamma.\gamma\,\{RL = \langle \alpha, \langle G_\gamma, \sigma_\gamma, N_\gamma, B_\gamma\rangle : (C'_\gamma : F'_\gamma)\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha \left\{\begin{array}{l} FL := \langle wait(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle \\ : (\langle redo(\gamma) : G_\alpha, \sigma_\alpha, N_\alpha, B_\alpha\rangle : F_\alpha)\end{array}\right\}, \\ \qquad\qquad \gamma\,\{RL := \langle \alpha, \langle fail : nil, \sigma_\gamma, N_\gamma, B_\gamma\rangle : (C'_\gamma : F'_\gamma)\rangle : R_\gamma\} \\ \Gamma \xrightarrow{\langle Redo_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$Redo_2$ — recognition of that backtracking of actor γ cannot be performed during execution of actor α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Subgoal = redo(\gamma)\} \\ \Gamma.\gamma\,\{RL = \langle \alpha, C_\gamma : \underline{nil}\rangle : R_\gamma\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle Redo_2, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

3.2 Interaction of Logical Actors

The abstract machine implements the following operations on this stage:

1. The comparison of substitutions that correspond to various actors of the program.
2. Neutralization of some actors.
3. Repeated proving of neutral actors.

$Check_1$ — checking if the actors of the program are consistent (during termination of proving of actor α).

Let us introduce some additional notions:

- $is\_neutral(\Gamma, \gamma) \stackrel{def}{=} \Gamma.\gamma\,\{Result = neutral\}$;
- $is\_active(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \ \wedge\ \Gamma.\gamma\,\{GL \neq success\}$;
- $is\_proven(\Gamma, \gamma) \stackrel{def}{=} \neg is\_neutral(\Gamma, \gamma) \ \wedge\ \Gamma.\gamma\,\{GL = success\}$;
- $SUBST(\Gamma, \gamma)$ is substitution σ_γ, $\Gamma.\gamma\,\{Subst = \sigma_\gamma\}$, or empty substitution ε, if $is\_neutral(\Gamma, \gamma)$;
- $does\_exist(\Gamma, \gamma) \stackrel{def}{=} \langle \gamma, M_\gamma, R_\gamma\rangle \in \Gamma$;
- $\Sigma(\Gamma, \{\alpha_1, \ldots, \alpha_n\}) = \bigcup_{i=1}^{n} SUBST(\Gamma, \alpha_i)$ — is a set of substitution assignments corresponding to all the actors $\alpha_1, \ldots, \alpha_n$ in state Γ.

Definition 1. *Set S of substitution assignments is conflicting one, if there are two subsets σ_1 and σ_2 and a variable X such that:*

1. *σ_1 and σ_2 are substitutions.*
2. *These substitutions gives values V_1 and V_2 to the X variable, that have no most general unifier.*

$$inconsistent(S) \stackrel{def}{=} \exists\, \sigma_1 \subset S \ \wedge\ \exists\, \sigma_2 \subset S \ \wedge\ \exists\, X : \neg\exists\, mgu(X\sigma_1, X\sigma_2).$$

Definition 2. $consistent(S) \stackrel{def}{=} \neg inconsistent(S)$ *— is a consistent set of substitution assignments.*

Definition 3. *A set of names NA of actors to be neutralized and proved repeatedly $may\_be\_neutralized(\Gamma, NA)$:*

1. $\forall \beta \in NA : does\_exist(\Gamma, \beta) \ \wedge\ is\_proven(\Gamma, \beta)$;
2. $\forall \beta \in NA :$
   $\exists$ *set of actors* $\{\alpha_i\}, i = 1, \ldots, k : does\_exist(\Gamma, \alpha_i) :$
   $inconsistent(\Sigma(\{\alpha_1, \ldots, \alpha_k, \beta\})) \ \wedge\ consistent(\Sigma(\{\alpha_1, \ldots, \alpha_k\}))$;
3. *A set of substitution equations of actors of any subset of Γ that has no common elements with the NA set should be consistent one.*

The condition (2) excludes any unnecessary neutralization of actors that are irrelevant to the contradictions that should be eliminated.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := neutralize(NA) : (restart(NA) : success)\} \\ \Gamma \xrightarrow{\langle Check_1, \alpha\rangle} \Gamma'\end{array}}$$

$Check_2$ — recognition of impossibility to eliminate contradictions between the actors with the help of neutralization of some actors (during termination of proving of actor α).

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = nil\} \\ \neg\exists\, NA : may\_be\_neutralized(\Gamma, NA)\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{GL := fail : nil\} \\ \Gamma \xrightarrow{\langle Check_2, \alpha\rangle} \Gamma'\end{array}}$$

$Neut_0$ — termination of neutralization of actors (during termination of proving of actor α).

$$\Gamma.\alpha\,\{GL = neutralize(\emptyset) : G\} \xrightarrow{\langle Neut_0, \alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

$Neut_1$ — neutralization of actor γ during execution of actor α:

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{Cont = \langle neutralize(\{\gamma\} \cup NA') : G, \sigma, N, B\rangle\},\ \gamma \notin NA' \\ \Gamma.\gamma\,\{RL = R\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha\,\{Cont := \langle neutralize(NA') : G, \sigma, [\gamma|N], B\rangle\}, \\ \qquad\qquad \gamma\,\{RL := neutral : R\} \\ \Gamma \xrightarrow{\langle Neut_1, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

$Succ$ — termination of proving of actor α with success.

$$\Gamma.\alpha\,\{GL = restart(\emptyset) : G\} \xrightarrow{\langle Succ, \alpha\rangle} \Gamma.\alpha\,\{GL := G\}$$

Call — invocation of repeated proving of actor γ during execution of α.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{FL = \langle restart(\{\gamma\} \cup RA') : G, \sigma, N, B\rangle : F\}\,,\ \gamma \notin RA' \\ \Gamma.\gamma = \langle \gamma, m(v_1, \ldots, v_k), R\rangle\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \alpha \left\{ \begin{array}{l} FL := \langle wait(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle \\ : (\langle redo(\gamma) : (restart(RA') : G)\,, \sigma, [\gamma|N], B\rangle : F) \end{array} \right\}, \\ \qquad\qquad \gamma\,\{RL := \langle \alpha, \langle m(v_1, \ldots, v_k) : nil, \underline{\varepsilon}, [\,], [\,]\rangle : nil\rangle : R\} \\ \Gamma \xrightarrow{\langle Call, \alpha, \gamma\rangle} \Gamma'\end{array}}$$

Note that the $Check_1$, the $Neut_1$, and the *Call* schemas make abstract machine nondeterministic one.

Con — resumption of proving of actor β after termination of proving of actor α that was invoked by actor β.

$$\frac{\begin{array}{l}\Gamma.\alpha\,\{GL = success\} \\ \Gamma.\beta\,\{GL = wait(\alpha) : G\}\end{array}}{\begin{array}{l}\Gamma' = \Gamma : \beta\,\{GL := G\} \\ \Gamma \xrightarrow{\langle Con, \beta, \alpha\rangle} \Gamma'\end{array}}$$

Note that defined abstract machine provides a possibility for modeling destructive assignment of variables with the help of logical actors. For instance, the $X := Y$ build-in predicate is implemented in the Actor Prolog language, that invokes the interaction between the actors of the program. The operational semantics of the *':='* predicate is straightforward one:

1. The predicate tries to unify the X and the Y terms.
2. If the most general unifier exists, the interaction of actors of the program is performed in accordance with the rules described above.
3. If neutralization and repeated proving of actors provides consistency between the actors of the program, the execution of the *':='* predicate terminates with success. In another case backtracking occurs in the program.

The model-theoretic semantics of this predicate is exactly the same as the semantics of the usual equality *'='* in pure Prolog and the operational semantics of the *'='* predicate is a special case of the *':='* predicate operational semantics.

4 Operational Semantics

The operational semantics of sequential logic program enhanced with logical actors is a map $\mathcal{O}$ that projects definition of procedures D and an initial state of program Γ^0, $\Gamma^0.\tau = \langle \tau, m(t_1, \ldots, t_k), R_\tau\rangle$, into the set of finite and infinite chains of states obtained with the help of transition schemas defined above.

Definition 4. *Operational semantics $\mathcal{O}$:*

$$\mathcal{O}[D, \Gamma^0] \stackrel{def}{=} \begin{array}{l} \left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{SUCCESS}\right\} \cup \\ \left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \Gamma_n^{FAILURE}\right\} \cup \\ \left\{\Gamma^0 \xrightarrow{l_1} \Gamma_1 \xrightarrow{l_2} \ldots\right\}. \end{array}$$

Note that the model-theoretic semantics of defined @-language strictly corresponds to the model-theoretic semantics of pure Prolog without negation.

Definition 5. *An initial set of actor constraints is a set of logical statements that corresponds to all the proven actors of initial state* Γ^0*:*

$$Init \stackrel{def}{=} \bigwedge_i M_i \text{ for all } \langle \alpha_i, M_i, R_i \rangle \in \Gamma^0, \text{such that } is\_proven(\Gamma^0, \alpha_i).$$

Proposition 1 (on soundness of the operational semantics). *The operational semantics* $\mathcal{O}$ *is sound, i.e., the success final state of the program can be obtained only if union of procedure definitions* D *with the negation of conjunction of initial set* $Init$ *and goal statement* $m(t_1, \ldots, t_k)$ *is unsatisfiable:*

$$\left(\Gamma^0 \xrightarrow{\star} \Gamma^{SUCCESS}\right) \Rightarrow \left(D \cup \{\neg\,(Init \ \wedge \ m(t_1, \ldots, t_k))\} \models \bot\right).$$

Proposition 2 (on completeness of the operational semantics). *The success final state of the program will be obtained if a substitution* θ *exists, such that*

$$D \models (Init \ \wedge \ m(t_1, \ldots, t_k))\,\theta,$$

and no infinite computations arise: $\Gamma^0 \xrightarrow{\star} \Gamma^{SUCCESS}$.

Thus, the program can fall into an infinite computation even if a success branch is present in the AND-OR tree, like the standard sequential Prolog.

Nevertheless the additional operation of neutralization of actors cannot provoke looping of the program, because the neutralization of active actors is prohibited in schema $Check_1$.

The practical use of the control strategy under consideration requires that the abstract machine stops after the obtaining of the first success final state despite the fact that the abstract machine can implement the exhaustive search until all existed answers are computed or an infinite computation occurs. This restriction corresponds to the perturbation model of constraint-based languages, i.e., the problem to be solved by the abstract machine is to fit given system of constraints to new information income from outer world only. After that, the abstract machine will wait for a new outside influence.

Conclusion

The logical actors concept gives an alternative to the nonmonotonic approach in logic programming. It forms a basis for solving the problem of ensuring soundness and completeness of the destructive assignment operation as well as strict classical model-theoretic semantics of logic programs operating in dynamic environment (such as graphical user interface and Internet).

The repeated proving of sub-goals allows to modify the logical reasoning during the execution of a logic program. Following the principle of modifiable

reasoning, we have developed concurrent object-oriented logic language Actor Prolog that ensures soundness of logic programs operating under conditions of permanent altering and updating of input information [11,8,12]. The ideas stated in this paper are approved by practical experiments with visual logic programming and Web agent logic programming [7].

The author is grateful to Prof. Yu.V. Obukhov, Dr. A.F. Polupanov, Dr. A.N. Kruglov, and Dr. S.V. Remizov (IRE RAS) for help and support in implementing the project, to Acad. Yu.I. Zhuravlev and Prof. V.A. Zakharov (Moscow State University) for fruitful discussions of the problem.

This work was supported by RFBR, project no. 06-07-89302.

References

1. Chesnevar, C.I., Maguitman, A.G., Loui, R.P.: Logical models of argument. ACM Computing Surveys **32**(4) (2000) 337–383
2. Alferes, J., Pereira, L.: Logic programming updating — a guided approach. In Kakas, A., Sadri, F., eds.: Computational Logic: From Logic Programming into the Future — Essays in honour of Robert Kowalski. Volume 2. Springer (2002) 382–412
3. Dix, J., Furbach, U., Niemelae, I.: Nonmonotonic reasoning: Towards efficient calculi and implementations. In Voronkov, Robinson, eds.: Handbook of Automated Reasoning. Volume 2. Elsevier (2001) 1121–1234
4. Eiter, T., Fink, M., Sabbatini, G., Tompits, H.: Using methods of declarative logic programming for intelligent information agents. Theory and Practice of Logic Programming **2**(6) (2002) 645–709
5. Rossi, F.: Constraint (Logic) Programming: A Survey on Research and Applications. In: New Trends in Constraints: Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 1999. Selected Papers. Volume 1865 of LNAI., Springer (1999) 40–74
6. Sannella, M., Maloney, J., Freeman-Benson, B., Borning, A.: Multi-way versus one-way constraints in user interfaces: Experience with the DeltaBlue algorithm. Software — Practice and Experience **23**(5) (1993) 529–566
7. Morozov, A.: Development and application of logical actors mathematical apparatus for logic programming of Web agents. In Palamidessi, C., ed.: ICLP 2003. LNCS 2916, Springer (2003) 494–495
8. Morozov, A., Obukhov, Y.: An approach to logic programming of intelligent agents for searching and recognizing information on the Internet. Pattern Recognition and Image Analysis **11**(3) (2001) 570–582 Available at: http://www.cplire.ru/Lab144/pria570m.pdf.
9. Morozov, A.: Visual logic programming based on the SADT diagrams. In Dahl, V., Niemela, I., eds.: ICLP 2007. LNCS 4670, Springer (2007) 436–437
10. Morozov, A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recognition and Image Analysis **13**(4) (2003) 640–649 Available at: http://www.cplire.ru/Lab144/pria640.pdf.
11. Morozov, A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In Sagonas, K., Tarau, P., eds.: Proc. of the IDL'99 Int. Workshop, Paris, France (1999) 39–53 Available at: http://www.cplire.ru/Lab144/paris.pdf.
12. Morozov, A.: Getting Started in Actor Prolog. IRE RAS (2002) Available at: http://www.cplire.ru/Lab144/start/.