On Semantic Link Between Logic, Object-Oriented, Functional and Constraint Programming *

Alexei A. Morozov

Institute of Radio Engineering and Electronics
Russian Academy of Sciences
ul. Mokhovaya 11, Moscow, 101999 Russia
morozov@mail.cplire.ru
http://www.cplire.ru/Lab144/index.html

Abstract. In the article I investigate semantic and operational links among object-oriented, logic, functional and constraint programming paradigms by an example of implementing them in logic language Actor Prolog. I show how means developed for ensuring model-theoretic semantics of object-oriented and functional features in the Actor Prolog have led to creation of a method of implementing a weak form of constraints in the logic language. These means have provided an interesting mathematical property of implemented constraints, namely, they guarantee that delaying computation and deadlocks never violate the completeness of a logic program.

Introduction

The main difference between the Actor Prolog [11,13,12,15,14] and other logic and constraint languages is in the control strategy that implements a modification of logic inference in the case of updating of input values during execution of a program. Development of this language and corresponding control strategies are necessary for solving the following problem. The standard control strategy of Prolog as well as its concurrent and distributed extensions (see, e.q., [7,17]) lose the soundness, if a program is executed in the dynamic environment. Foregoing control strategies do not support defeasance of results of computation when the input data of the program are changed. Therefore the developers of the languages were forced either to renounce explicitly the classical model-theoretic semantics of the language and use some non-classical logic systems, or simply ignore the problem. At the same time mathematically rigorous logic programming of the systems that operate in the dynamic environment, such as Internet agents, interactive systems, virtual reality systems, etc. is impossible without a solution of this problem. One can use standard Prolog for programming intelligent agents,

* This work was supported by the Russian Foundation for Basic Research, projects no. 00-01-00560 and 01-01-06278.

but definitely it would not be the logic programming in mathematically rigorous sense.

Our solution of the problem is based on the principle of repeated proving subgoals of a logic program [12,15,14]. Logic program is divided into separate subgoals (logical actors), linked by common variables. The control strategy of Actor Prolog automatically reveals contradictions between the current state of environment and the results of the logic inference. Revealed contradictions are eliminated by repeated proving of the actors, that (and only that) lose their soundness because of the contradictions. Thus, our approach radically differs from the non-monotonic ones, because Actor Prolog is based on the classical first order logic; the modification of reasoning is implemented with the help of the new control strategy, but not by the refusal of the classical logic.

Usage of logical actors in a logic language leads to a new programming paradigm that in some sense turns inside out usual constraint programming paradigm. Whereas a usual constraint language delays the computation if there are no enough input data, Actor Prolog always computes some solution corresponding to the model-theoretic semantics of a program. Then, Actor Prolog modifies the logic inference and all the results, if some new data appear [14]. In other words Actor Prolog guarantees that delayed computations and deadlocks never violate completeness of the logic program, if there are no infinite loops in it. Thus, Actor Prolog actually implements a sound and exhaustive search.

In this article I will restrict myself to considering computation in one separate logical actor. The detailed consideration of logical actors one can find in [12,15,14]. Our approach to interpreting object-oriented paradigm in the logic programming is discussed in [12].

It is interesting that whereas an implementation of logical actors does not require any use of delayed computations we have faced this concept during creation of object-oriented means of Actor Prolog. In the section 1 a detailed arguing is presented of the use of restricted form of delayed computations in Actor Prolog. Our major motivation for the restriction of delaying was keeping the completeness of logic inference, but the further investigation of this idea has led to a new composition of existing engineering ideas and principles of object-oriented, functional and constraint programming, that were used in other projects of recent decade. It is shown in the section 2 that the restricted use of delaying computation provides a new declarative and operational semantics for sending messages to data items.

This idea along with the functions discussed in the section 3 has led to a method of implementation of a weak form of constraints in the logic language, considered in the section 4. These means have provided required mathematical property of implemented constraints, namely, a constraint logic program has such model-theoretic semantics that the program is sound and complete w.r.t. this model-theoretic semantics (if the program has no infinite loops), even if delayed subgoals or deadlocks are present. In the section 5 Actor Prolog is compared with other multi-paradigm logic languages.

1 Semantics of Sending Messages to Unbound Variables

Sending a message to an object (call the method of the object) is an ordinary operation of imperative object-oriented languages:

$$Object.method(A, B, \ldots, C)$$

In the context of logic programming this operation is interesting for us, because it is not defined for the unbound values of the argument *Object*. Actually, it is not clear, what a computer should do, if the variable *Object* is unbound? There are several possible answers to this question. And these different answers lead to creation of different programming styles.

1. Some programming languages simply raise a real-time error in this situation. This solution is not interesting for us, because it ignores the declarative semantics of a program and is not a mathematical one.
2. One can eliminate the problem by imposing certain restrictions on the syntax of logic language (these restrictions are to be checked during the translation). We used this approach in the early versions of Actor Prolog. However this approach reduces the expressiveness of the language. For example, in early versions of the language the use of instances of classes in the arguments of predicates was prohibited. Now we have refused this approach.
3. In imperative languages one can easily solve the problem with the help of initializing all variables in the program by some default values. The value of any variable can be updated by the operation of destructive assignment later. Unfortunately, this approach is unacceptable for logic languages, because unbound variables are very important feature of the logic programming and refusal of their usage would essentially reduce descriptive capabilities of logic language.
4. Another solution is based on the idea of non-deterministic substitution of objects existing in a program into the operation of message sending [3]. Thus, the operation of message sending becomes non-deterministic in general case. Backtracking of non-deterministic program will dispatch the message to all objects of the program. From our point of view, this approach is not satisfactory, because it means sending the messages to all objects of the program, including irrelevant ones. Moreover, this approach does not work, if the objects are created dynamically, because in this case the full list of possible receivers for a message is unknown during the execution of the program.
5. A radical solution of the problem is full refusal of the convenient understanding of objects and messages. This approach is used in so-called "process view" to implementation of objects in logic programming [3]. The idea of this approach is in the following: the "objects" are simulated with the help of concurrent recursive procedures and "messages" are simulated with the help of binding of infinite lists of common variables of these predicates (this approach corresponds to the stream parallelism [5]). Thus, in such languages the operator of message sending "." is not used at all. We think that this

approach is not satisfactory, because it is oriented rather towards the simulation of interacting objects, instead of being oriented towards supporting object style of programming. In particular, we believe that the process view badly reflects the structural and dynamic aspects of OOP [12].

6. Yet another solution is to use co-routining and delay to ensure that methods are only dispatched when the *Object* is sufficiently instantiated. From our point of view this approach is too cumbersome like previous one, because it treats all objects in the program as separate processes. Moreover, it is not clear what is the model-theoretic semantics of a system of these concurrent processes.
7. Our solution of the problem is based on the idea of delayed subgoals without co-routining.

In the Actor Prolog the operations of message sending are denoted as

$$World \ ? \ message(A, B, \dots, C)$$

The operation will be delayed, if the variable $World$ is not bound. Then the program continues its work according to the standard control strategy. If the variable $World$ is bound, the delayed operation will invoked. If invoked operation completes successfully, the execution of the program will continue. In another case the standard backtracking occurs in the logic program.

Delaying subgoals is widely used in functional logic languages [6,7] as well as in constraint systems [10,17]. Our approach differs from other ones in the following: *we use delayed subgoals only for supporting model-theoretic semantics of the operation of message sending in the language.* In other words, it is only one variable in the operation of message sending that could delay this subgoal. This variable is the "target" variable $World$ designating the receiver of the message. This restricted use of delaying subgoals keeps not only the soundness, but also the completeness of control strategy of the language according to the theorem 1 considered below.

We will not consider the problem of implementation of classes and objects in this article, because it is possible to use any existing method for this purpose. For example, one can use standard label/argument or label/predicate approach [3]. The following theorem is independent of implementation of classes and objects in the language.

Theorem 1. *Let us consider a logic program written in the pure Prolog with standard control strategy. The objects are implemented in the program with the help of standard label/argument or label/predicate method [3]. Let us assume also that interpreter can delay the execution of message passing, if the variable denoting target object is unbound. So, if the program has no infinite loops, our control strategy ensures the soundness and completeness of logic inference.*

The sketch of the proof. (1) Let us assume that the program was completed and there are no delayed subgoals after its termination. In this case the soundness and completeness of the inference is gave by the general theorem on soundness and completeness of SLD-resolutions [9]. (2) Let us assume that there are some delayed subgoals after the termination of the program. In this case one can assign a dummy object *Dummy* to the target variables in all the delayed subgoals. The object *Dummy* should contain dummy facts corresponding to all predicates of the program, so the execution of any predicate will be successfully completed in this object without any binding of the arguments. This assignment will cause an invocation of all delayed subgoals. The assignment will cause no inconsistencies among the subgoals of the program, because only the target variables can delay subgoals in our language. The delayed subgoals will be invoked and terminated with success. So, we will get the case (1) immediately.

Note, that the completeness of the inference in the theorem means that one could use the language for implementing exhaustive search, i.e. the program could find *all* existing solutions of a problem.

2 What Is the Semantics of Sending Messages to Terms?

It is no surprise that the notions “instance of class” and “data item” have different semantics in the framework of logic programming. This difference is caused by so-called problem of object identity.

The problem of object identity is that the developer of a language should decide, what is the criterion to distinguish an object from every other one? There are two possible answers to this question:

1. One can consider two objects as identical ones if they have the same content. For instance, we could consider that two arrays $(1, 2, 3)$ and $(1, 2, 3)$ are identical, because they have the same items and these items are placed in the same order.
2. The notion “object” being used in practical programming have slightly more complex semantics. For example, let us assume two objects “Window”. Let us assume that these objects have identical attributes “color”, “title”, “text” and so on. But these objects corresponds to different elements of graphic user interface. So, they are different objects, though their elements are identical.

The former approach is most convenient for development of object-oriented logic languages and is widely used in the field of deductive databases [2,4]. As a matter of fact, it corresponds to classical understanding of the objects as generalized data items. In the framework of this approach one can define the operation of unification on the compound objects. For example, the unification of arrays $(1, 2, C)$ and $(A, 2, 3)$ will cause unification of their elements; the variables A and C will be instantiated by values 1 and 3 respectively. Moreover, the unification will make these objects identical; they will be the same instance of the class “Array”.

The second approach is closer to the real life, however, it causes some theoretical problems. Namely, the notions “object” (an instance of a class) and “data item” will be different ones. It means, that we could not introduce operation of unification on these "objects". For instance, we cannon "unify" two graphic windows on the screen, because they actually are different objects even they contain the same image.

In Actor Prolog data items (simple and compound terms) are not instances of classes, and instances of classes are not terms. Therefore, two compound terms $[1, 2, 3]$ and $[A, B, C]$ could be unified in Actor Prolog, but two instances of the class 'Window' could not. An attempt to unify two variables bound with the different instances of a class (or, especially, of some different classes) will be always failed. Each operation of creation of the instance of a class results in producing a new unique instance of the class. This instance could not be unified with any other object or term. By the way, it is the reason why we name instances of classes as the "worlds", but not as the "objects" in Actor Prolog.

Having accepted a concept that a data item is not an instance of class, we developed a new interpretation of the operation of message sending

$$World\ ?\ message(A, B, \ldots, C)$$

for the cases, when the variable $World$ contains not a world, but a data item (term). In Actor Prolog we have proposed the following approach (we called it as "placing the target into the list of arguments").

Definition 1 (The method "placing the target into the list of arguments"). *In the case if the variable $World$ contains a term, the predicate 'message' is called in the same world, when the current clause is executed (one can designate this world as '$self$'), and the term $World$ is added to the list of arguments of the predicate:*

$$self\ ?\ message(World, A, B, \ldots, C)$$

For example, expression $A\ ?\ message(B)$, where A is a data item, denotes a call of the predicate $message(A, B)$ in Actor Prolog.

Theorem 2. *The syntactical transformations of logic programs (keeping their operational semantics) simulating usage of the method "placing the target into the list of arguments" during the execution of programs do exist.*

The sketch of the proof. Let us add auxiliary predicates to logic program, corresponding to operations of message sending. For every expression $W\ ?\ p(A, B, \ldots, C)$ one should create two clauses

```
call_p(W,A,B,...,C):-
     is_world(W),
     W ? p(A,B,...,C).
call_p(W,A,B,...,C):-
     is_data(W),
     p(W,A,B,...,C).
```

where *is_world* and *is_data* are auxiliary predicates, that check if the variable W contains a world or a data item respectively. One can eliminate the case of free variable W with the help of method of delaying subgoals described in the section 1. Let us replace all operations of message sending by calls of corresponding predicates $call\_p(W, A, B, \ldots, C)$. So, we will obtain the program with the same operational semantics.

The theorem 2 allows us to define the model-theoretic semantics of a logic language implementing the method "placing the target into the list of arguments".

Definition 2. *The model-theoretic semantics of a logic program P written in the pure Prolog extended by classes and objects in the sense of the theorem 1, placing the targets into the lists of arguments, is the model-theoretic semantics of the program P' obtained from the program P with the help of transformations in the sense of the theorem 2.*

It is interesting that the separation of concepts "instance of class" and "data item" provides a solution for one another well-known problem of logic languages. The problem is that sending of a message to an object is an asymmetric operator by its nature; the arguments of the operator (receiver of the message and the predicate) play very different roles during the execution of the operator. At the same time, the uniform use of arguments in the operations is more convenient for the logic. For instance, the arguments A and B have the equal status in the relation $A > B$. So, any attempt to read such an expression in the object-oriented style — A receives message '$> B$' — looks a bit unnatural. Our interpretation of the operation of message sending resolves this problem. For example, the expression $A\ ?\ '<'\ (B)$ in the Actor Prolog, where A is a number, denotes a call of usual predicate $'<'\ (A, B)$ in the current search space '$self$'.

3 Arithmetic Operators Are Not Constructors of Compound Terms in Actor Prolog

Actor Prolog allows a use of the following atomic formulas in the head of a clause:

$$f(A, B, \ldots, C) = R$$

This syntactical construction is similar to equations in some functional logic languages [6]. However, their have another semantics. Namely, a clause with such head in Actor Prolog can be called in two different ways:

1. It can be called as a usual predicate $f(A, B, \ldots, C)$. In this case the variable R is not used and the construction "$= R$" is ignored.
2. It can be called as a function $?f(A, B, \ldots, C)$. In this case the variable R denotes the value returned by the function. Note that the prefix "?" is used to distinguish calls of functions and compound terms.

As a matter of fact, described syntactic construction defines two predicates at once:

1. A predicate $f(A, B, \ldots, C)$.
2. An auxiliary predicate $f'(R, A, B, \ldots, C)$ with additional argument R [1].

One can use calls of functions in clauses of a program in any place when a term could be used. For example, the following atomic formula is allowed in the language:

$$p(1, 2, 3, ?f(7, 8, 9))$$

Here the call of the function f is used as an argument of the predicate p. The standard technique of flattening [6] is used for translating such programs. So, the expression under consideration will be transformed into the list of two subgoals:

```
f'(Result,7,8,9),
p(1,2,3,Result).
```

where *Result* is a unique variable denoting the value to be returned by f.

Let us consider the canonical example of usage of functions in a logic language, namely, a definition of the function *append*. One can define this function in the Actor Prolog in the following way:

```
append([],L) = L.
append([H|L1],L2) = [H|?append(L1,L2)].
```

During the translation of the program this definition will be transformed into the following clauses[2]:

```
append'(L,[],L).
append'(R0,[H|L1],L2):-
    append'(R1,L1,L2),
    R0 == [H|R1].
```

Theorem 3. *The program written in the pure Prolog extended by the syntactic means of Actor Prolog for implementing functions has standard model-theoretic semantics.*

The sketch of the proof. The theorem follows from the definition of considered syntactic constructions.

The means of functional logic programming under consideration have helped us to decide yet another well-known problem of logic programming. The problem is that the operators $+$, $-$, $*$, etc. are conveniently used in the standard Prolog for defining compound terms (structures), but not for denoting calls of subroutines. For example, expression $A+B$ in the standard Prolog denotes the term $'+'(A, B)$. Thus, the use of usual arithmetic expressions in the standard Prolog is rather inconvenient. For instance, one should use two subgoals and special built-in predicate *is* in the standard Prolog to implement an ordinary call of the predicate p with the numeric argument equal to $A + B$:

[1] Let us denote the names of auxiliary predicates by the apostrophes.
[2] Note that the predicate $=$ of standard Prolog is denoted as $==$ in Actor Prolog.

```
C is A + B,
p(C).
```

In the Actor Prolog we have developed another semantics for the standard arithmetic operators. Namely, the compiler converts all arithmetic expressions into the calls of functions. For instance, Actor Prolog will "understand" the expression under consideration $p(A + B)$ and will interpret it as the following list of subgoals:

```
R == ?'+'(A,B),
p(R).
```

Certainly, we have implemented a set of built-in arithmetic functions '+', '-', '*', *sin*, *cos* etc. in Actor Prolog instead of the predicate *is* of standard Prolog.

I will not discuss the implementation of second order functions in the Actor Prolog in this article. These features are based on the use of so-called underdetermined sets [12] and some simple means for parameterization of predicates and functions [13].

4 An Example of Logic Program with Constraints

Now I will consider an example of constraint logic program written in Actor Prolog (the full text of the program is presented in the appendix). The goal of the program is to distribute an amount of money to defined entries of budget according to bookkeeping rules and some wishes of the user.

Built-in arithmetic predicates of Actor Prolog cannot operate with unbound arguments (just as the built-in predicates of the standard Prolog cannot do it). So, I will implement the constraints in the logic program on the base of considered object-oriented and functional features of the language. We have developed a special technique for implementing logical rules "understanding" any combinations of bound and unbound arguments in the head.

One can explain the idea of this technique considering the following example. Let us implement constraint "the sum" $sum(A, B, C)$ using the features of Actor Prolog:

```
sum(A,B,C):-
     C == A ? sum_ix(B),
     C == B ? sum_ix(A),
     B == C ? sub_ix(A).
```

The first subgoal in the rule is a call of auxiliary function *sum_ix* in the object A. The operational semantics of the language considered in the previous sections guarantees that the actual call of the function *sum_ix* will be performed only when the variable A has a bound value. Moreover, variable A could have only numerical values in this example, so the first subgoal (according to the method "placing the target into the list of arguments" considered in the section 2) denotes the call of function with two arguments $sum\_ix(A, B)$ in the world $self$. According to the semantics of the Actor Prolog, this function will be implemented as a predicate with three arguments $sum\_ix'(C, A, B)$.

All three subgoals in the body of the rule *sum* have the same declarative semantics, but their possible delay is caused by different variables A, B, C. Thus, if any one of the arguments of the predicate *sum* gets a bound value it causes a call of an subgoal in the clause.

The predicate $sum\_ix(A, B) = C$ is defined on the base of the same principle. The only difference is that the first argument of it always will be bound (in another case the subgoal is delayed). Therefore we need only two subgoals in the body of the rule:

```
sum_ix(A,B) = C :-
     C == B ? sum_ii(A),
     B == C ? sub_ii(A).
```

Similarly, the auxiliary predicate $sum\_ii(A, B) = C$ needs only one subgoal in the corresponding rule. And the structure of the program guarantees that the arguments A and B of this predicate will be bound. Therefore one can safely use the built-in function in this rule:

```
sum_ii(A,B) = A + B.
```

There are some additional predicates and functions defined in the appendix, namely, "substraction", "multiplication" and "is more or equal". They are implemented on the base of the same technique as the predicate *sum*.

The function $natural\_number(Min, Quantity)$ checks if a natural number belongs to defined interval (or generates numbers belonging to this interval). The argument Min denotes the lower bound of the interval and the variable $Quantity$ denotes the quantity of natural numbers in the interval.

Let us consider the goal statement of the program now. There are two input values of the program: a total sum of money $Total$ and the cost of computers to be bought $NewHardware$.

```
Total == 30000.0
NewHardware == 10000.0
```

The goal of the program is to calculate the following entries of the budget: "the cash money" $Cash$, "the charges" $Overhead$, "the taxes" Tax and "the equipment" $Equipment$. There are the following additional rules for calculating these values:

1. The total sum of money includes "overhead" and "useful charges" $sum(Useful, Overhead, Total)$.
2. The charges are equal to 10 % of the total sum $mult(Total, 0.1, Overhead)$.
3. The salary includes the cash money and the taxes $sum(Cash, Tax, Salary)$.
4. The taxes are equal to 35.8 % of the cash money $mult(Cash, 0.358, Tax)$.

5. The useful charges includes the salary and the money for buying new equipment $sum(Equipment, Salary, Useful)$.
6. The sum intended for buying new equipment should not be less than the sum required for the buying new computers
$ge(Equipment, NewHardware)$.
7. The sum for acquisition of new equipment should never exceed 20 % of the cost of new hardware $ge(NewHardware * 1.2, Equipment)$.
8. The sum obtained in cash should be a round one
$Cash\ ==\ ?natural\_number(1, Total/100.0) * 100.0$.

The program has calculated a number of solutions of the problem. Here is one of existing fifteen solutions:

```
Total = 30000.0   Overhead  = 3000.0     Tax = 4296.0
Cash  = 12000.0   Equipment = 10704.0
```

There are no unbound variables among the solutions, so the theorem 1 guarantees that the program has calculated all the solutions. All the calculated solutions are sound w.r.t. the model-theoretic semantics of the program. Thus, the example under consideration is a sound and *complete* constraint logic program. Note that the property of completeness is certainly an achievement for a constraint programming language.

5 Comparison with Other Approaches

Our approach to functional logic programming is simpler than many others implemented earlier (see survey [6]). The goal of developing this approach was in creating a sound and complete control strategy for the functional logic language and increasing descriptive power of the language, rather than increasing efficiency or computational speed. Note also that functions in Actor Prolog are not means for implementing parallel computations, objects or featured terms, as against to some other approaches discussed below.

A constraint functional logic language OZ [17] implements functions (by adding an extra argument) and delays of messages like Actor Prolog. The main difference of Actor Prolog is that its control strategy is complete in the sense of the theorem 1. Moreover, Actor Prolog implements repeated proving subgoals of the logic program [12,15,14]. So, one can use Actor Prolog for mathematically correct logic programming of Internet agents and other applications operating in the dynamic environment. Unfortunately, there is no distributed implementation of Actor Prolog as against to OZ. Note also that Actor Prolog implements the method “placing the target into the list of arguments”. The functions in Actor Prolog can be used both as functions and as predicates.

The functions in Visual Prolog [18] can be used as predicates, however Visual Prolog does not allow the use of unbound variables as objects receiving messages.

It is interesting to compare the combination of delays and non-deterministic functions implemented in Actor Prolog with the computing model of functional logic language Curry [7], that uses residuation and narrowing. One can say that flattening of non-deterministic functions in combination with the standard control strategy in Actor Prolog is a simplest case of narrowing from the implementation point of view. As a result, the control strategy of Actor Prolog is complete in the sense of the theorem 1. The interaction with the environment is implemented in Curry on the basis of monads, as against to Actor Prolog that implements the concept of logical actors. Moreover, we support the convenient interpretation of object-oriented notions “class” and “instance of class”, while the objects in the object-oriented extension of Curry (named ObjectCurry [8]) are simulated with the help of stream parallelism.

The composition of flattening and delaying subgoals is used in functional extension NUE of logic language NU-Prolog [16] as well as in Actor Prolog. However, functions in NUE-Prolog can be only deterministic ones as against to Actor Prolog. The execution of a function in NUE-Prolog is delayed if there are some unbound arguments in the head of a rule. This restriction was imposed on NUE-Prolog, in particular, for implementing concurrent execution of functions. Concurrency in Actor Prolog is implemented on another level of abstraction (processes are a kind of instances of classes [14]), therefore there is no need in such restriction in Actor Prolog.

All functions should be deterministic in the language LIFE [1] too. The execution of a function will be delayed if there are unbound arguments. It is interesting, that Actor Prolog have a feature named “underdetermined sets”, that is similar to so-called ψ-terms of LIFE. However we do not consider the underdetermined sets as an implementation of “objects” in Actor Prolog as against to LIFE. One can find a comparison of underdetermined sets with ψ-terms in [12].

In general our implementation of classes and objects corresponds to the standard “clauses view” to the logic OOP [3], but we have extended this approach by restricted delaying of message sending and the method “placing the target into the list of arguments”.

Conclusions

We have considered a subset of the control strategy of Actor Prolog, namely, the computation of one separate logical actor. The restricted use of delaying subgoals has led to a new composition of existing engineering ideas and principles of object-oriented, functional and constraint programming, that were used in other projects of recent decade. In particular, we have proposed the method “placing the target into the list of arguments” providing new declarative and operational semantics for sending messages to data items. In the composition with a method of implementing functions in Actor Prolog it has led to creation of a new method of implementation of a weak form of constraints in a logic language. These means have provided interesting mathematical property of implemented constraints, namely, a constraint logic program has such model-theoretic semantics that the program is sound and complete w.r.t. this model-theoretic semantics (if the program has no infinite loops), even if delaying subgoals or deadlocks are present.

I am grateful to Prof. Yuri V. Obukhov, Dr. Alexander F. Polupanov and all my colleagues, friends and relatives for their help and support. I am thankful also to the anonymous referees for their comments and suggestions.

References

1. H. Aït-Kaci, B. Dumant, R. Meyer, A. Podelski, and P. van Roy. *The Wild LIFE Handbook.* Digital Equipment Corporation, prepublication edition, March 1994. (http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/impl/fp_lp/life/0.html).
2. S. Ceri, G. Gottlob, and L. Tanka. *Logic Programming and Databases.* Springer, 1990.
3. Andrew Davison. A survey of logic programming-based object-oriented languages. In P. Wegner, A. Yonezawa, and G. Agha, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 42–106. MIT Press, 1993. (http://www.cs.mu.oz.au/tr_db/mu_92_03.ps.gz).
4. D. W. Embley, S. W. Liddle, and Y.-K. Ng. On harmonically combining active, object-oriented, and deductive databases. In *ADBIS'96: Proc. of the Third Intern. Workshop on Advances in Databases and Information Systems*, pages 21–30, Moscow, Russia, 1996. MEPhI. (http://citeseer.nj.nec.com/9806.html).
5. Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of Prolog programs: a survey. *Programming Languages and Systems*, 23(4):472–602, 2001. (http://citeseer.nj.nec.com/311564.html).
6. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994. (http://www.informatik.uni-kiel.de/~mh/publications/papers/JLP94.html).
7. M. Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93, 1997. (http://www.informatik.uni-kiel.de/~mh/publications/papers/POPL97.html).
8. M. Hanus, F. Huch, and P. Niederau. An object-oriented extension of the declarative multi-paradigm language Curry. In *Proc. of the 12th International Workshop on Implementation of Functional Languages (IFL 2000)*, number 2011 in LNCS, pages 89–106. Springer, 2001. (http://www.informatik.uni-kiel.de/~mh/publications/papers/IFL00.html).
9. G. Hogger. *Introduction to Logic Programming.* Academic Press, 1988.
10. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994. (http://citeseer.nj.nec.com/334926.html).
11. Alexei A. Morozov. Actor Prolog. *Programmirovanie*, (5):66–78, 1994. In Russian.
12. Alexei A. Morozov. Actor Prolog: an object-oriented language with the classical declarative semantics. In K. Sagonas and P. Tarau, editors, *Proceedings of the International Workshop on Implementation of Declarative Languages (IDL'99)*, pages 39–53, Paris, France, September 1999. (http://www.cplire.ru/Lab144/paris.pdf).
13. Alexei A. Morozov and Yuri V. Obukhov. Actor Prolog. definition of programming language. Preprint 2(613) of 14.06.96, IRE RAS, Moscow, Russia, June 1996. In Russian. (http://www.cplire.ru/Lab144/index.html).

14. Alexei A. Morozov and Yuri V. Obukhov. An approach to logic programming of intelligent agents for searching and recognizing information on the Internet. *Pattern Recognition and Image Analysis*, 11(3):570–582, 2001. (http://www.cplire.ru/Lab144/pria570m.pdf).
15. Alexei A. Morozov and Yuri V. Obukhov. On the problem of logical recognition in the dynamic Internet environment. *Pattern Recognition and Image Analysis*, 11(2):454–457, 2001. Proceedings of the 5th International Conference "Pattern Recognition and Image Analysis: new Information Technologies" PRIA. Samara, Russia, 16–22 October 2000 (http://www.cplire.ru/Lab144/pria5.pdf).
16. Lee Naish. Adding equations to NU-Prolog. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 15–26, Passau, Germany, August 1991. Springer. (http://www.cs.mu.oz.au/~lee/papers/eq/).
17. P. van Roy. Logic programming in Oz with Mozart. In *Proc. of the International Conference on Logic Programming*, pages 38–51. MIT Press, 1999. (ftp://ftp.ps.uni-sb.de/pub/papers/Others/lpinoz99.ps.gz).
18. *Visual Prolog Version 5.0. Language Tutorial.* Prolog Development Center, Copenhagen, Denmark, 1997. (http://www.pdc.dk).

A An Example of Constraint Logic Program

```
project: (('Main'))
class 'Main' specializing 'Alpha':
con = ('Console');
[
goal:-
     Total == 30000.0,
     NewHardware == 10000.0,
     --
     sum(Useful,Overhead,Total),
     mult(Total,0.1,Overhead),
     sum(Cash,Tax,Salary),
     mult(Cash,0.358,Tax),
     sum(Equipment,Salary,Useful),
     ge(Equipment,NewHardware),
     ge(NewHardware*1.2,Equipment),
     Cash == ?natural_number(1,Total/100.0)*100.0,
     --
     con ? writeln("Total = ",Total),
     con ? writeln("Cash = ",Cash),
     con ? writeln("Overhead = ",Overhead),
     con ? writeln("Equipment = ",Equipment),
     con ? writeln("Tax = ",Tax,"\n"),
     --
     fail.
goal:-
     con ? writeln("No more solutions.").
-- The predicate "Sum"
sum(Ax,Bx,Cx):-
     Cx == Ax ? sum_ix(Bx),
     Cx == Bx ? sum_ix(Ax),
     Bx == Cx ? sub_ix(Ax).
sum_ix(Ai,Bx) = Cx :-
     Cx == Bx ? sum_ii(Ai),
     Bx == Cx ? sub_ii(Ai).
sum_ii(Ai,Bi) = Ai + Bi.
-- The function "Subtraction"
sub_ix(Ai,Bx) = Cx :-
     BN == Bx ? '-'(),
     Cx == BN ? sum_ii(Ai),
     BN == Cx ? sub_ii(Ai),
     Bx == BN ? '-'().
sub_ii(Ai,Bi) = Ai - Bi.
-- The predicate "Multiplication"
mult(Ax,Bx,Cx):-
     Cx == Ax ? mult_ix(Bx),
     Cx == Bx ? mult_ix(Ax),
     Ax == Cx ? div_ix(Bx).
mult_ix(Ai,Bx) = Cx:-
     Cx == Bx ? mult_ii(Ai),
     Bx == Cx ? div_ii(Ai).
mult_ii(Ai,Bi) = Ai * Bi.
-- The function "Division"
div_ix(Ai,Bx) = Cx :-
     BI == Bx ? inv(),
     Cx == BI ? mult_ii(Ai),
     CI == Cx ? inv(),
     Bx == CI ? mult_ii(Ai).
div_ii(Ai,Bi) = Ai / Bi :-
     Bi <> 0.0.
inv(Ai) = 1 / Ai :- Ai <> 0.0.
-- The predicate "Is more or equal"
ge(Ax,Bx):-
     Ax ? ge_ix(Bx).
ge_ix(Ai,Bx):-
     Bx ? '<' (Ai).
-- The function "A natural number belongs to the interval"
natural_number(Min,_) = Min.
natural_number(Min,Quantity)
     = 1 + ?natural_number(Min,Quantity-1) :-
     Quantity > 1.
]
```