Mathematical Problems of Computer Science 24, 2005, 34–41.

On Interpreters of Logic Programming Systems

Semyon A. Nigiyan and Aram M. Hambardzumyan

Department of System Programming, Yerevan State University,
e-mail nigiyan@ysu.am, me76@front.ru

Abstract

We introduce the notions of totally resolving and totally complete interpreters for Horn programming languages. We prove the existence of totally complete interpreter (an interpreter which gives all the answers for a query if the query is a logical consequence of the program) for any Horn programming language and existence of totally resolving interpreter (an interpreter which gives all the answers for any program and query) for languages whose programs have finite templates of their least models. We also consider problems of total completeness and total resolvability for PROLOG interpreter from viewpoint of some (natural) program transformations and prove that it is not possible to make the interpreter totally complete.

1 Introduction

In this paper, we discuss questions concerning the extension of logic (Horn) programming systems' capabilities. We focus our attention on the problem of total completeness, that is the ability of the interpreter to answer queries which are logical consequences of the programs, and if such queries contain variables then to find all the possible values for them (in contrast to simple logical completeness (see [1]) which means finding at least one tuple of values for variables), and total resolvability, that is being totally complete and being able to answer all the queries which are not logical consequences of the programs (in contrast to simple resolvability (see [1]) which means being logically complete and answering all the queries which are not logical consequences of the programs). We prove the existence of totally complete interpreter for logic programming languages and the existence of totally resolving interpreters for languages whose programs have finite templates of their least models (the notion of program least model's template has been defined in [2]). Also, we study possibilities to make PROLOG interpreter [3] totally complete by means of program transformations. It is known (see [4, 5]) that PROLOG interpreter is logically correct (i. e. doesn't lie), but isn't logically complete. In [1] truncated simple monadic PROLOG has been considered (i. e. PROLOG which doesn't use built-in predicates, doesn't use predicates of predicates, uses functional symbols of 0 arity and predicate symbols of 0 and 1 arity only, and whose program clause bodies' lengths do not exceed 1 and the queries' lengths equal 1). It has been shown that, by permuting and removing clauses of PROLOG programs, it's impossible to lead the interpreter to logically complete even for this version of PROLOG, but if, in addition, the programs have no more than one repeated predicate symbol in the clause heads then it is possible to make the interpreter resolving. Also, in [6] it has been shown that by permuting and removing clauses of PROLOG programs and permuting atoms in queries and clause bodies, one can make the interpreter logically complete for simple monadic PROLOG whose programs use no more than one repeated predicate symbol in heads of their clauses (i. e. when the lengths of queries and program clause bodies may exceed 1) but cannot make it resolving. We prove that by permuting and removing clauses of PROLOG programs, one cannot make the interpreter totally complete for truncated simple monadic PROLOG whose programs use no more than one repeated predicate symbol in heads of their clauses, and hence cannot make it totally resolving.

Definitions and Results

Let's fix three countable disjoint sets X, F, Π. X is a set of object variables, F is a set of functional symbols such that each symbol from F is assigned some arity and for any $n \geq 0$ the subset of n-ary functional symbols is countable, and Π is a set of predicate symbols such that each symbol from Π is assigned some arity and for any $n \geq 0$ the subset of n-ary predicate symbols is countable. Each variable from X and each 0-ary functional symbol from F are terms. If f is n-ary functional symbol from F and $t_1, \dots, t_n$ are terms, $n \geq 1$ then the construction of form $f(t_1, \dots, t_n)$ is also a term. Each 0-ary predicate symbol from Π is an atom. If p is n-ary predicate symbol from Π and $t_1, \dots, t_n$ are terms, $n \geq 1$, then the construction of form $p(t_1, \dots, t_n)$, is also an atom. The formula of first order predicates logic uses elements of sets $X, F, \Pi; \neg, \vee, \&, \supset$ logic operations; and $\forall, \exists$ quantifiers, and is defined in the ordinary way. Let's denote the set of all closed formulas by Φ.

Let's describe the interpretations we are interested in. The object set of these interpretations is the set M of terms without variables (ground terms). Each 0-ary functional symbol from F is associated with itself. For each n-ary ($n \geq 1$) functional symbol $f \in F$ there corresponds a mapping $M^n \to M$ that associates a tuple $< t_1, \dots, t_n >$ with a term $f(t_1, \dots, t_n)$. Each 0-ary predicate symbol from Π is associated with one of elements of set $\{true, false\}$, and for each n-ary ($n \geq 1$) predicate symbol $p \in \Pi$ there corresponds some mapping $M^n \to \{true, false\}$. Let's denote the set of these interpretations by H.

Let $A, B \in \Phi$. We'll say that formula B is a logical consequence of formula A and denote this fact by $A \models B$ if the value of the formula $A \supset B$ is true for any interpretation of the set H.

Let $\Phi' \subset \Phi$ and $\Phi' \neq \emptyset$. A pair (A, A'), where $A, A' \in \Phi'$, will be called the transformation of the formulas of Φ'.

Let T be some non-empty subset of transformations of formulas of the set Φ'. We will say that the formula $A \in \Phi'$ is T-transformable into the formula $B \in \Phi'$ (this fact will be denoted by $A \xrightarrow{T} B$) if a sequence of transformations $(A_1, A_2), \dots, (A_{n-1}, A_n)$ from T exists such that $A_1 = A, A_n = B$, where $n > 1$. If $A, B \in \Phi'$ and $A \xrightarrow{T} B$, the formula B will be called the T-image of A.

Let's define the set $\mathcal{P}$ of logic programs. The program $P \in \mathcal{P}$ is a sequence of clauses $S_1, \dots, S_n, n \geq 0$. A clause S from P is either a fact A or a rule $A : -B_1, \dots, B_m$ ($m > 0$) that is an implication $B_1 \& \dots \& B_m \supset A$, where $A, B_1, \dots, B_m$ are atoms. A is called the head of the clause S, the sequence $B_1, \dots, B_m$ is called the body, and m is called the length of the clause body. Program P is associated with the formula $\forall x_1, \dots, \forall x_r (S_1 \& \dots \& S_n)$ where $x_1, \dots, x_r$ are all the variables used in $S_1, \dots, S_n, r \geq 0$.

Let's define the set $\mathcal{Q}$ of queries. The query $Q \in \mathcal{Q}$ has the form $? - C_1, \dots, C_k$

where $C_1,\dots,C_k$ are atoms, $k \geq 1$. The query Q is associated with the formula $\exists y_1,\dots,\exists y_s(C_1 \& \dots \& C_k)$ where $y_1,\dots,y_s$ are all the variables used in $C_1,\dots,C_k, s \geq 0$; k is called the length of the query Q. The set $\{y_1,\dots,y_s\}$ is denoted by $Var(Q)$.

The logic programming language is defined as a pair $< Prog, Quer >$, where $Prog \subseteq \mathcal{P}$ and $Quer \subseteq \mathcal{Q}$. We'll denote the language $< \mathcal{P}, \mathcal{Q} >$ by $\mathcal{L}$.

Let $< Prog, Quer >$ be a logic programming language. For any $P \in Prog$ and any $Q \in Quer$, the set of answers corresponding to logical semantics is denoted $Log(P,Q)$ and is defined as follows:

if $P \not\models Q$ then $Log(P,Q) = \{no\}$;

if $P \models Q$ and $Var(Q) = \emptyset$ then $Log(P,Q) = \{yes\}$;

if $P \models Q$ and $Var(Q) = \{y_1,\dots,y_s\}, s > 0$ then $Log(P,Q) = \{< t_1,\dots,t_s > | t_1,\dots,t_s \in M$, and if Q' is a result of replacing y_1 with $t_1,\dots,y_s$ with t_s in Q, then $P \models Q'\}$.

The interpreter U for the language $< Prog, Quer >$ is an algorithm which, for every $P \in Prog, Q \in Quer$, produces a set $U(P,Q)$ which is defined as follows:

if $Log(P,Q) \subset \{yes, no\}$ then $U(P,Q) \subseteq Log(P,Q)$;

otherwise $U(P,Q) \subseteq \{< t_1,\dots,t_s > | t_1,\dots,t_s$ are terms, $s > 0$, $Var(Q) = \{y_1,\dots,y_s\}$, and if $< \tau_1,\dots,\tau_s >$ is a result of replacing all the variables of $< t_1,\dots,t_s >$ with terms from M then $< \tau_1,\dots,\tau_s > \in Log(P,Q)\}$.

For interpreter U and $P \in Prog, Q \in Quer$, we define following set $Ans(U,P,Q)$:

if $U(P,Q) \subset \{yes, no\}$ then $Ans(U,P,Q) = U(P,Q)$;

otherwise $Ans(U,P,Q) = \{< \tau_1,\dots,\tau_s > | \ \tau_1,\dots,\tau_s \in M$ and there exists such a tuple $< t_1,\dots,t_s > \in U(P,Q)$ that $< \tau_1,\dots,\tau_s >$ can be obtained from $< t_1,\dots,t_s >$ by replacing all its variables with some terms from $M\}$.

If $P \models Q \Rightarrow Ans(U,P,Q) = Log(P,Q)$ for any $P \in Prog$ and any $Q \in Quer$ then the interpreter U is called totally complete. If $Ans(U,P,Q) = Log(P,Q)$ for any $P \in Prog$ and any $Q \in Quer$ then the interpreter U is called totally resolving.

A finite set $\sigma = \{t_1/y_1,\dots,t_n/y_n\}, n > 0$, is called a substitution if $y_1,\dots,y_n$ are variables, $t_1,\dots,t_n$ are terms, $i \neq j \Rightarrow y_i \neq y_j, 1 \leq i,j \leq n$, and $t_i \neq y_i, 1 \leq i \leq n$. If $n = 0$ then σ is called empty substitution. The application of substitution σ to an atom results in simultaneous replacement of variables $y_1,\dots,y_n$ in that atom with terms $t_1,\dots,t_n$ respectively. Composition of substitutions is defined in a natural way (see [4]).

Atoms A and B are called unifiable if there exists such substitution σ (which is called a unifier of A and B) that $A\sigma = B\sigma$. A unifier σ of A and B is called the most general unifier and is denoted by $mgu(A,B)$ if for any unifier δ of A and B there exists such a substitution γ that $\sigma\gamma = \delta$. For any pair of unifiable atoms there exists most general unifier (see [4]).

Theorem 1 *There exists totally complete interpreter for the language $\mathcal{L}$.*

Proof. Let ρ be SLD resolution rule with selection function which selects first atom from the query. Then by applying ρ to non-empty query Q: $? - C_1,\dots,C_k$ $(k > 0)$ and clause S: $A : -B_1,\dots,B_m$ $(m \geq 0)$ we obtain query Q': $? - B_1\sigma,\dots,B_m\sigma, C_2\sigma,\dots,C_k\sigma$ where $\sigma = mgu(A, C_1)$ (see [4]). We denote this fact by $\rho(Q,S) = Q'$.

Let P be a program and Q be a non-empty query. Then the sequence of queries $Q_1,\dots,Q_n$ $(n \geq 1)$ is called inference of Q_n from (P,Q) if $Q_1 = Q$ and $Q_{i+1} = \rho(Q_i, S_{j_i})$ where $S_{j_i} \in P, i = 1,\dots,n-1$. This fact is denoted by $(P,Q) \vdash Q_n$.

Let's define the answer set $Proc(P,Q)$ which corresponds to procedural semantics:

if $(P,Q) \not\vdash ?-$, then $Proc(P,Q) = \{no\}$;

I $(P,Q) \vdash ?-$ and $Var(Q) = \emptyset$, then $Proc(P,Q) = \{yes\}$;
f $(P,Q) \vdash ?-$ and $Var(Q) = \{y_1, \ldots, y_s\}$ $(s > 0)$, then $Proc(P,Q) = \{< t_1, \ldots, t_s > \in$
there exist such an inference $Q_1, \ldots, Q_n$ of empty query from (P,Q) (i. e. $Q_1 = Q$
$Q_n = ?-$) and such substitution δ that $\{t_1/y_1, \ldots, t_s/y_s\} \subseteq \sigma_1 \ldots \sigma_{n-1}\delta$, where σ_i is the
stitution which corresponds to the application of ρ to Q_i and some clause from P resulting
Q_{i+1} $(1 \le i < n)\}$.

The proof of Theorem 1 results from the fact that $Proc(P,Q) = Log(P,Q)$, which follows
n [4].

Theorem 1 is proved.

In order to formulate and prove Theorem 3, let's introduce following notions which have
n defined in [2, 4].

The interpretation $I \in H$ is called a model of a program P if the formula which is
ociated with P is true on I. It is known that every program P has the least model I_P
().

We say that atom A precedes atom B (and we denote it by $A \prec B$) if there exists such
ubstitution σ that $A\sigma = B$ (for convenience reasons, we will presume that σ does not
tain items like t/x where x is not included in A). One can notice that the preceding
ation is reflexive and transitive.

We say that atom A is congruent with atom B (and denote it by $A \equiv B$) if $A \prec B$ and
$\prec A$. One can notice that the congruence relation is reflexive and transitive.

It follows from the results of [2] that the relations of preceding and congruence are
idable.

We say that the set $\mathbf{A_1}$ of atoms is congruent to set $\mathbf{A_2}$ of atoms (and denote it by
$\equiv \mathbf{A_2}$) if there exists such one-to-one mapping $\varphi : \mathbf{A_1} \rightarrow \mathbf{A_2}$ that $A \equiv \varphi(A)$ for every
om $A \in \mathbf{A_1}$.

Let's define the contraction of $\mathbf{A}$ (which we denote by $\mathbf{A}^{con}$):

$\mathbf{A}^{con} \subset \mathbf{A}$;

$A \in \mathbf{A}^{con}, \mathrm{B} \in \mathbf{A}^{con}$ and $A \prec B \Rightarrow A = B$;

$A \in \mathbf{A} \Rightarrow$ there exists such $B \in \mathbf{A}^{con}$ that $B \prec A$.

It is easy to see that any two contractions of $\mathbf{A}$ are congruent, and if $\mathbf{A_1}$ and $\mathbf{A_2}$ are
om sets then $\mathbf{A_1} \equiv \mathbf{A_2} \Rightarrow \mathbf{A_1^{con}} \equiv \mathbf{A_2^{con}}$.

Let P be a program. For every $i \ge 1$ we introduce the notion of i-subtemplate of the
ast model of P (i-subtemplate of I_P for short). $K_P^1 = Facts(P)^{con}$ is 1-subtemplate of I_P,
here $Facts(P)$ is the set of facts of the program P. Let $i \ge 1$ and K_P^i be i-subtemplate of
. In order to give the definition of $i+1$-subtemplate K_P^{i+1} we need to define following set
$\tilde{K}_P^{i+1}$:

$A \in \tilde{K}_P^{i+1} \Leftrightarrow$there exists a rule $S \in P$ of the form $B : -B_1, \ldots, B_m$ and an atom sequence
$A_1, \ldots, A_m$, such that:

atoms $A_1, \ldots, A_m$ don't share variables with each other and with S;

for every $j = 1, ..., m$ an atom $A'_j \in K_P^i$ exists such that $A_j \equiv A'_j$;

there exist such substitutions $\sigma_1, \ldots, \sigma_m$ that $\sigma_1 = mgu(A_1, B_1), \sigma_2 =$
$mgu(A_2, B_2\sigma_1), ..., \sigma_m = mgu(A_m, B_m\sigma_1 \ldots \sigma_{m-1})$ and $A = B\sigma_1 \ldots \sigma_m$.

$$K_P^{i+1} = (K_P^i \cup \tilde{K}_P^{i+1})^{con}$$

It's easy to see that for every $i \ge 1$ any two i-subtemplates of I_P are congru...

Let's ... we define the following ...

$A \in \bar{K}_P \Leftrightarrow$ there exists such $i_0 \geq 1$ that for every $i \geq i_0$ such an atom $A_i \in K_P^i$ exists that $A \equiv A_i$. The template of I_P is defined as $\bar{K}_P^{con}$ and is denoted by K_P. It's easy to see that any two templates of I_P are congruent.

According to [2], for any program P and ground atom A_0 we have:

$$A_0 \in I_P \Leftrightarrow \text{there exists such an atom } A \in K_P \text{ that } A \prec A_0.$$

Let $? - C_1, \ldots, C_k$ be a query ($k > 0$) and $Var(? - C_1, \ldots, C_k) = \{y_1, \ldots, y_s\}, s > 0$. According to [4],

$$Log(P, ? - C_1, \ldots, C_k) = \{< t_1, \ldots, t_s > \in M^s \,| C_i\{t_1 / y_1, \ldots, t_s / y_s\} \in I_P, 1 \leq i \leq k\}.$$

Lemma 2 *For finite templates of the least model, the problem of their construction is decidable.*

Proof. The algorithm of template's construction follows directly from its definition. Let's prove that it doesn't run infinitely. First, we need to show that the sequence of subtemplates contains items which are congruent to each other and that they are also congruent to the template of the program's least model.

It follows from the finiteness of K_P that there exists such $j_0 \geq 1$ that for any atom $A \in K_P$ and any $j \geq j_0$ there exists such an atom $B \in K_P^j$ that $A \equiv B$. On the other hand, for any $j \geq j_0$ and any atom $A' \in K_P^j$ there exists such an atom $B' \in K_P$ that $B' \prec A'$ (it follows from how K_P^i and K_P are constructed). For $i \geq j_0$, there exists an atom $B'' \in K_P^i$, $B'' \equiv B'$. Since K_P^i is a contraction, $B'' = A'$, and hence $B' \equiv A'$. Thus, there exists $j_0 \geq 1$ such that:

for any atom $A \in K_P$ and for any $j \geq j_0$, there exists such an atom $B \in K_P^j$ that $A \equiv B$;

for any $j \geq j_0$ and any atom $A' \in K_P^j$, there exists such an atom $B' \in K_P$ that $B' \equiv A'$.

Basing on these results and taking into account the fact that K_P^j are K_P contractions, we have that $K_P^j \equiv K_P, j \geq j_0$, and hence $K_P^{j_0} \equiv K_P^{j_0+1}$.

So, we proved that there exists such $j_0 \geq 1$ that $K_P^{j_0} \equiv K_P^{j_0+1}$.

On the other hand, if $K_P^i \equiv K_P^{i+1}$ for some $i \geq 1$, then $K_P^i \equiv K_P^{i+1} \equiv K_P^{i+2} \equiv \ldots$, and hence, $K_P \equiv K_P^i \equiv K_P^{i+1} \equiv K_P^{i+2} \equiv \ldots$. So, we have:

$$K_P^i \equiv K_P^{i+1} \Rightarrow K_P \equiv K_P^i, i \geq 1.$$

The algorithm constructing K_P sequentially builds subtemplates until it reaches such j that $K_P^j \equiv K_P^{j+1}$. To prove that it doesn't run infinitely, let's describe the algorithm of contraction construction for finite set of atoms and show that it doesn't run infinitely.

Algorithm "Contraction".

Input: finite set of atoms $\mathbf{A}$.

Output: $\mathbf{A}^{con}$.

Step 1. If there exist such atoms $A, B \in \mathbf{A}, \mathbf{A} \neq \mathbf{B}$, that $A \prec B$ (decidability of precedence relation follows from [2]) then remove $B \in A$ and go to step 1, otherwise halt and give $\mathbf{A}$ as an output.

It's easy to see that given algorithm builds contraction of the input set. It follows from the finiteness of the input set that the algorithm cannot run infinitely.

To prove that K_P construction algorithm doesn't run infinitely, we have to show that every subtemplate is constructed in finite time.

$K_P^1 = Facts(P)^{con} \Rightarrow$ (using "Contraction" algorithm) K_P^1 can be constructed in finite time. Supposing that K_P^i is constructed in finite time for $i \geq 1$, let's construct K_P^{i+1}. $K_P^{i+1} = (K_P^i \cup \bar{K}_P^{i+1})^{con}$. It's easy to see that $\bar{K}_P^{i+1}$ can be constructed in finite time. On the other hand, the contraction $(K_P^i \cup \bar{K}_P^{i+1})^{con}$ can also be constructed with *"Contraction"* algorithm in finite time. Hence, subtemplate K_P^{i+1} also can be constructed in finite time.

So, each subtemplate can be built in finite time, and one of them is congruent with the template of the least model. Therefore, the template of the least model also can be constructed in finite time.

Lemma 2 is proved.

Theorem 3 *There exists a totally resolving interpreter for languages whose programs have finite templates of their least models.*

Proof. Let's describe the algorithm of such interpreter.

Input. Query Q: $?-C_1, \ldots, C_k$ $(k > 0)$ and a program P.

Step 1. Build the template of I_P (according to Lemma 2, this process will end in finite time). Let's denote the template by K_P.

Step 2. Build a program P' which consists of the facts using all the atoms form K_P (in arbitrary sequence). It's easy to notice that K_P is a template for $I_{P'}$, and according to [2] $P' \overset{\Delta}{\approx} P$.

Step 3. Build the inference tree for P' and Q using SLD-resolution rule ρ. The tree's depth is limited with the amount of atoms in the source query, and the amount of immediate descendants of each node is limited with the amount of clauses of P'. Therefore, the tree is finite.

Step 4. If none of the leaves contains an empty query then output 'no' and halt. Otherwise if $Var(Q) = \emptyset$ and at least one of the leaves contains empty query then output 'yes' and halt. Otherwise if $Var(Q) = \{y_1, \ldots, y_s\}$ $(s \geq 1)$ then halt and give out the set $\{< t_1, \ldots, t_s > \in M^s \mid$ the tree contains an inference $Q_1, \ldots, Q_n$ of empty query from (P, Q) (i. e. $Q_1 = Q, Q_n = ?-$) for which such a substitution δ exists that $\{t_1/y_1, \ldots, t_s/y_s\} \subseteq \sigma_1 \ldots \sigma_{n-1}\delta$, where σ_i is the substitution being used to derive Q_{i+1} from Q_i, $(1 \leq i < n)\}$ (since the inference tree is finite, this set will also be finite).

This description shows that the algorithm will halt in finite time for any input, and hence it is a totally resolving interpreter's algorithm.

Theorem 3 is proved.

Corollary of Theorem 3. There exists a totally resolving interpreter for the language which doesn't use functional symbols with arity > 0.

Let $< Prog, Quer >$ be some logic programming language, and let $P_1, P_2 \in Prog$. We say that P_1 and P_2 are Δ-equivalent (denoted by $P_1 \overset{\Delta}{\approx} P_2$) if $P_1 \models Q \Leftrightarrow P_2 \models Q$ for any $Q \in Quer$.

Having a programming language $< Prog, Quer >$ and some set of transformations T of programs from $Prog$, we will say that it is possible to make the interpreter of the language totally complete (accordingly, totally resolving) using transformations from T if for any P

from $Prog$ there exists such $P' \in Prog, P' \overset{\Delta}{\approx} P, P \overset{T}{\to} P'$ that $P \models Q \Rightarrow Ans(U, P', Q) = Log(P', Q)$ (accordingly, $Ans(U, P', Q) = Log(P', Q)$) for any $Q \in Quer$.

We will say that the program P contains the repeated predicate symbol p if the number of the clauses of P, the heads of which use p, is greater than one.

Let's define the sets $Prog$ and $Quer$ for truncated simple monadic PROLOG with one repeated predicate symbol. The programs and queries of $< Prog, Quer >$ use 0-ary functional symbols and 0- and 1-ary predicate symbols. The lengths of program's clause bodies do not exceed 1, and the lengths of queries equal 1. Any program from $Prog$ uses no more than one repeated predicate symbol. PROLOG interpreter is defined in [3].

Let T be the set of transformations permuting and removing program clauses so that the resulting program is Δ-equivalent to the initial one.

Theorem 4 *Using transformations from T, one cannot make the interpreter of truncated simple monadic PROLOG with one repeated predicate symbol totally complete.*

Proof. Let's consider the following program P and query Q.

The program P:

$$p(a)$$
$$p(b) : -p(x)$$
$$p(c) : -p(x)$$

The query Q: $? - p(x)$.

$Log(P, Q) = \{a, b, c\}$. It's obvious that any T-image of P is some permutation of P (it can't be a subsequence since any proper subsequence of P is not Δ-equivalent to P). It's easy to see that for any permutation P' of P, $Ans(U, P, Q) \neq Log(P, Q)$.

Thus, the theorem 4 is proved.

Corollary of Theorem 4. Using transformations from T, one cannot make the interpreter of truncated simple monadic PROLOG with one repeated predicate symbol totally resolving.

References

[1] Nigiyan S. A., Khachoyan L. O. *Transformations of Logic Programs*. Programming and Computer Software, Vol. 23, No. 6, pp. 302-309, 1997.

[2] Nigiyan S. A, Khachoyan L. O. *On Δ-equivalence problem of logic programs*. Reports of National Academy of Sciences of Armenia, Vol. 99, No. 2, pp. 99-103 (in Russian), 1999.

[3] Clocksin W. F., Mellish C. S. *Programming in Prolog*. Berlin: Springer-Verlag, 1984.

[4] Lloyd J. W. *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1984.

[5] Nigiyan S. A. *The Prolog Interpreter from the Viewpoint of Logical Semantics*. Programming and Computer Software, Vol. 20, No. 2, pp. 69-75, 1994.

[6] Hambardzumyan A. M. *The Completeness and Solvability Problems for Simple Monadic PROLOG Interpreter*. Proceedings of the Conference on Computer Science and Information Technologies, Yerevan, pp. 36-38, 1999.

Տրամաբանական ծրագրավորման համակարգերի ինտերպրետատորների մասին

Ս. Ա. Նիգիյան, Ա. Մ. Համբարձումյան

Ամփոփում

Աշխատանքում ներկայացվում են տոտալ լուծելի և տոտալ լրիվ ինտերպրետատորների հասկացությունները Հորնի ծրագրավորման լեզուների համար: Ապացուցվում է տոտալ լրիվ ինտերպրետատորի (ինտերպրետատոր, որը տալիս է հարցման բոլոր պատասխանները, եթե հարցումը հանդիսանում է ծրագրի տրամաբանական հետևանք) գոյությունը՝ կամայական Հորնի ծրագրավորման լեզվի համար, և տոտալ լուծելի ինտերպրետատորի (ինտերպրետատոր, որը կամայական ծրագրի և հարցման համար տալիս է բոլոր պատասխանները) գոյությունը այն լեզուների համար, որոնց ծրագրերի փոքրագույն մոդելների շաբլոնները վերջավոր են: Նաև դիտարկվում են տոտալ լրիվության և տոտալ լուծելիության հարցերը ՊՐՈԼՈ-ի ինտերպրետատորի համար՝ ծրագրերի և հարցումների որոշ (բնական) ձևափոխությունների տեսակետից, և ապացուցվում է, որ ինտերպրետատորը հնարավոր չէ դարձնել տոտալ լուծելի: