On Computability by Logic Programs

Andrei Voronkov **Uppsala University**[1]

Abstract. The problem of computational completeness of Horn clause logic programs is revisited. The standard results on representability of all computable predicates by Horn clause logic programs are not related to the real universe on which logic programs operate. SLD-resolution, which is the main mechanism to execute logic programs, may give answer substitutions with variables. As the main result we prove that computability by Horn clause logic programs is equivalent to standard computability over the Herbrand universe with variables. The semantics we use is *S-semantics* introduced by Falaschi e.a. [3]. As an application of the main result we prove the existence of a metainterpreter for a sublanguage of real Prolog, written in the language of Horn clauses with the S-semantics. We also show that the traditional semantics of Prolog do not reflect its computational behavior from the viewpoint of recursion theory.

1 Introduction

The statement that Horn clause programs can compute all computable predicates has become folklore in the logic programming literature. However the statement about computational completeness of Horn clauses does not state that Horn clauses can compute all functions which are intuitively computable on the domain of discourse of Prolog. Whereas Horn clauses operate on the set of all terms in some finite language, the completeness results are proved for universes containing only ground terms. In [9, 12] the completeness is stated for the universe constructed from the constant 0 by repeated applications of the constructor s. In a more advanced paper [1] the completeness is stated for all finite signatures generating an infinite the Herbrand universe, but still without variables.

Horn clauses programs, however, operate on arbitrary terms, which can in particular contain variables. The behavior of Horn clauses on such arbitrary terms is complicated due to the use of unification and most general unifiers in the standard operational semantics (SLD-resolution). Thus, data types handled by Horn clauses are not only ground terms. It was noted elsewhere that the characterization of logic programs in terms of ground queries is not adequate to its operational semantics [3]. In that paper several semantics were developed so as to capture the inherent property of Horn clauses to compute substitutions.

It is known that some aspects of semantics of Horn clause logic programs, for example negation, depend on particular properties of the Herbrand universe [10]. The structure of the Herbrand universe is also important from the computational viewpoint. The proofs of computational completeness of Horn clauses from e.g. [12, 1, 9] cannot be simply modified for the case where the language contains no constants.

The main problem we investigate in this paper is the Turing completeness of Horn clauses on arbitrary Herbrand universes generated by finite signatures. While it is easy to define what are computable functions or predicates on such a universe, it is not clear if one can compute them by Horn clause programs. Thus there is a gap between the standard result on procedural completeness of Horn clauses and the real domains on which logic programs operate. In this paper we prove computational completeness of Horn clauses for Herbrand universes with variables.

Consider for example the following set of terms:

$$\{f(t_1, t_2) \mid t_2 \text{ is obtained from } t_1 \text{ by renaming variables}\}.$$

This set *is* intuitively computable. But *all* the classical results on computability by Horn clauses [12, 1, 9] say us nothing about representability of this set by a Horn clause program.

Even if we prove that all computable predicates on natural numbers can be simulated by a suitable Horn clause logic program, it does not immediately follow that this is also true for computable predicates over the domain of discourse of Prolog. There are examples of a different behavior of notions of computability on different representations of natural numbers. For example, Minsky's machines [7] simulate all computable functions on a non-standard representation of natural numbers, but do not compute all computable functions on the standard representation. More complicated examples are numerous in the theory of enumerations [2].

In [3] two kinds of semantics to capture domains with variables have been defined: one corresponds to the arbitrary Herbrand universes (S-semantics in terms of [3]) and another one to the so called *upward closed* Herbrand universes (C-semantics). Our techniques allows one to handle both kinds of semantics. However some applications from section 3 are true only for S-semantics. To give a taste what these semantics are, we introduce some definitions.

We assume that we have a *finite* set F of function symbols with associated arities. Constants are considered as 0-ary functional symbols. Let $V = v_0, v_1 \ldots$ be an infinite countable set of variables.

We assume that the reader is familiar with the basic notions of substitutions, renamings and of most general unifiers.

The set *Term* of all terms in the alphabet $\langle F, V\rangle$ is defined as follows:

1. Each variable $v_i \in V$ belongs to *Term*;
2. If f is an $n-$ary function symbol from F, $t_1, \ldots, t_n \in$ *Term*, then $f(t_1, \ldots, t_n) \in$ *Term*.

Via Term^n we denote the set of all tuples $\langle t_1, \ldots, t_n\rangle$ with $t_i \in$ *Term*.

A set $S \subseteq \mathit{Term}^n$ is *well-formed* iff it is closed under renaming of variables.

We consider semantics of programs in terms of the *computed answer substitutions* [3]. There are at least two possibilities for such a definition. One corresponds to the arbitrary Herbrand universe with variables, and another one corresponds to the *upward-closed* Herbrand universe. They may be characterized as follows. For each Horn clause program $\mathcal{P}$ and each predicate symbol P we introduce two sets:

$$S_{\mathcal{P},P} = \{\langle t_1, \ldots, t_n\rangle \mid \text{For the query } P(x_1, \ldots, x_n) \text{ to the program } \mathcal{P} \text{ SLD-resolution can give an answer substitution } [t_1/x_1, \ldots, t_n/x_n]\}$$

$$C_{\mathcal{P},P} = \{\langle t_1,\ldots,t_n\rangle \mid \text{For the query } P(t_1,\ldots,t_n) \text{ to the program } \mathcal{P} \text{ SLD-resolution gives the empty answer substitution } []\}$$

The relation between the two semantics can be characterized by the following lemma:

Lemma 1.1 *Let $\langle t_1,\ldots,t_n\rangle$ be a tuple of terms. Then $\langle t_1,\ldots,t_n\rangle \in C_{\mathcal{P},P}$ iff there exists a tuple of terms $\langle s_1,\ldots,s_n\rangle \in S_{\mathcal{P},P}$ and a substitution θ such that $\langle s_1,\ldots,s_n\rangle\theta = \langle t_1,\ldots,t_n\rangle$.* ■

It is possible to characterize $S_{\mathcal{P},P}$ in terms of the positive hyperresolution [4]. Such a characterization can help in understanding the proofs from this paper.

Lemma 1.2 *Let $\mathcal{P}$ be a Horn clause program. Then the set $S_{\mathcal{P},P}$ can be defined by the following inductive definitions:*

1. *Each fact from the program $\mathcal{P}$ belongs to $S_{\mathcal{P},P}$.*
2. *The set $S_{\mathcal{P},P}$ is closed under renaming of variables.*
3. *Let atomic formulas $A_1,\ldots,A_n$ belong to $S_{\mathcal{P},P}$ and there is a rule $C\colon -B_1,\ldots,B_n$ in $\mathcal{P}$such that all variables from A_i are distinct from the variables of A_j and distinct from the variables of $C\colon -B_1,\ldots,B_n$. Let θ be the most general unifier of $A_1,\ldots,A_n$ with $B_1,\ldots,B_n$. Then $C\theta$ belongs to $S_{\mathcal{P},P}$.* ■

Let us give some examples.

In what follows we make the following agreement about the syntax. When displaying concrete Horn clause programs, the variable names start from the upper-case letters, while the other symbols start from the lower-case letters. However in all other parts of the text we denote variables by x, y, z, u, v, w, possibly with indices.

Example 1.1 *Below is the program computing the set of pairs*

$$\{\langle t_1, t_2\rangle \mid t_1, t_2 \text{ are two distinct variables}\} :$$

```
distinct_variables(X,Y).
```

Example 1.2 *Let the set F consist of the constant* `c` *and a binary function symbol* `g`*. The following program computes the set of terms, in which each variable has at most one occurrence:*

```
one_occurrence(X).
one_occurrence(c).
one_occurrence(g(X,Y)) :-
   one_occurrence(X),
   one_occurrence(Y).
```

Example 1.3 *Let the set F be as in the previous example. The following program computes the set of terms, in which one variable has exactly two occurrences, while all other variables have at most one occurrence:*

```
two_occurrences(T) :-
   two(X,T).

two(X,g(Y,Z)) :-
   one(X,Y),
   one(X,Z).
two(X,g(Y,Z)) :-
   zero(X,Y),
   two(X,Z).
two(X,g(Y,Z)) :-
   two(X,Y),
   zero(X,Z).

one(X,X).
one(X,g(Y,Z)) :-
   one(X,Y),
   zero(X,Z).
one(X,g(Y,Z)) :-
   zero(X,Y),
   one(X,Z).

zero(X,Y).
zero(X,c).
zero(X,g(Y,Z)) :-
   zero(X,Y),
   zero(X,Z).
```

These examples show one important feature of S-semantics: the set of computed answers $S_{\mathcal{P},P}$ is not closed under substitution (one can easily show this for each of the three programs). The last example also shows that it can be complicated even to express quite simple sets of terms in S-semantics.

2 Main result

We do not assume that the reader is familiar with the theory of recursive functions. The only fact that we use about computability is the characterization of recursively enumerable (computable) predicates given by Matiyasevič [6] (Theorem 2.2 below).

To speak about computability on abstract structures like $Term^n$, one of the possible approaches is to use the notion of *enumeration* [2]. We encode elements of $Term^n$ by natural numbers. A set of terms is considered computable iff the set of all codes for this set is recursively enumerable (i.e. computable on natural numbers). Let us fix some Gödel numbering γ of $Term^n$ in the following way.

Let $F = \{f_1, \ldots, f_n\}$. Let $p_0, p_1 \ldots$ be the sequence of all prime numbers in increasing order (such that $p_0 = 2, p_1 = 3, \ldots$ etc). Then[2]

$$\gamma(v_i) = 2^i$$

$$\gamma(f_i(t_1, \ldots, t_m)) = 3^i 5^{\gamma(t_1)} \times \ldots \times p_{m+1}^{\gamma(t_m)}$$

Definition 2.1 *A set* $S \subseteq Term^n$ *is* computable *iff the set*

$$\{\langle \gamma(t_1), \ldots, \gamma(t_n) \rangle \mid \langle t_1, \ldots, t_n \rangle \in S\}$$

is recursively enumerable. A relation R *is* computable *if such is the set* $\{\langle t_1, \ldots, t_n \rangle \mid R(t_1, \ldots, t_n)\}$. ■

Now we shall restrict our attention to the S-semantics. At the end of this section we will show how to modify the definitions and the proofs so as to include the C-semantics.

Definition 2.2 *A well-formed set* $S \subseteq Term^n$ *is* H-computable *(Horn clause computable) iff* $S = S_{\mathcal{P},P}$ *for a Horn clause program* $\mathcal{P}$ *and a predicate symbol* P. *A relation* R *is* H-computable *if such is the set* $\{\langle t_1, \ldots, t_n \rangle \mid R(t_1, \ldots, t_n)\}$. ■

The main aim of this paper is to show the equivalence of the two notions of computability (Theorem 2.5). This theorem states:

Let $Term^n$ be the set of all n-tuples of terms in a finite signature. Then a well-formed set $S \in Term^n$ is computable iff it is H-computable.

It may seem at the first sight that the proof of this statement can easily be obtained from known proofs by using the following "theorem on constants": a formula $\varphi(x)$ is provable from a set T of closed formulas iff $\varphi(a)$ is provable from T, where a is a fresh constant. But this observation does not work: an infinite number of constants is needed to represent some computable sets of terms, since the number of different variables in this set may be infinite. The classical result on computability of all recursively enumerable predicates in the signature $0, s$ is straightforward:: we prove it in Theorem 2.2 in half a page.

The proof of our main result requires complicated numerations based on the idea similar to that of "difference lists" in logic programming. This section is devoted to the proof of this main result. The proof is quite technical and the reader not interested in technical details may skip the rest of this section.

The rest of this section is structured as follows. In section 2.1 we consider the most simple case when the language F consists of constants only. In section 2.2 we show how to simulate natural numbers and recursive functions on natural numbers, if F has at least one non-constant. In section 2.3 we consider the case when F contains a function symbol of arity ≥ 2. In this case we can encode lists and substitutions, and prove the main statement using this encoding. In section 2.4 the case of the universe in which all function symbols are unary is considered. In this case we cannot encode all substitutions. However it is sufficient to represent substitutions of a fixed length, which proves the main statement for this case.

[2]The mapping γ defined here is not an enumeration in the sense of [2], but it does not matter for our purposes.

2.1 Universe without functions

We start from a simple special case of universes consisting only of constants and variables.

Theorem 2.1 *Let the set F of function symbols consist of only null-ary symbols, i.e. constants. Then any computable well-formed set $S \subseteq Term^n$ is H-computable.*

Proof. It is easy to see that $Term^n$ has only finite number of elements up to renaming variables. Thus we can represent them by a Horn clause program consisting only of facts. ■

2.2 Encoding natural numbers

From now on we assume that F contains a function symbol with arity ≥ 1. We fix one such symbol f.

We introduce special pairs of terms called *numerals* to encode natural numbers. For a natural number n the corresponding numeral will be denoted by $\widehat{n}$. More precisely, numerals are equivalence classes of pairs of terms w.r.t. renaming, but for simplicity we shall sometimes use them as just pairs of terms. The definition is as follows:

$$\widehat{0} = \langle v, v \rangle, \text{where } v \in V$$

$$\text{If } \langle t_1, t_2 \rangle = \widehat{n}, \text{then } \langle f(t_1, \ldots, t_1), t_2 \rangle = \widehat{n+1}$$

Let for a numeral $\langle t_1, t_2 \rangle$ the natural number it encodes be denoted by $\overline{t1, t2}$.

We denote via ω the tuple of terms $\langle v, f(v, \ldots, v) \rangle$, where v is a variable. The reason to introduce ω and to use such a representation of natural numbers is the following:

Lemma 2.1 *Let $\langle s_1, s_2 \rangle$ and $\langle t_1, t_2 \rangle$ be numerals.*

1. *The tuples $\langle s_1, s_2 \rangle$ and $\langle t_1, t_2 \rangle$ are unifiable iff $\overline{s_1, s_2} = \overline{t_1, t_2}$. In this case their most general unifier is a renaming substitution.*

2. *The tuple $\langle s_1, s_2 \rangle$ is not unifiable with ω.*

Proof. By routine inspection of the definition of numerals and ω using induction on $\overline{s_1, s_2}$. ■

Lemma 2.2 *The relation numeral$(t_1, t_2) \rightleftharpoons$ "the pair of terms $\langle t_1, t_2 \rangle$ is a numeral" is H-computable.*

Proof. It is easy to see that the following program computes the relation:

```
numeral(X,X).
numeral(f(X,...,X),Y) :-
   numeral(X,Y). ■
```

Lemma 2.3 *The following relations are H-computable:*

1. *zero$(t_1, t_2) \rightleftharpoons \overline{t_1, t_2} = 0$.*

2. $suc(s_1, s_2, t_1, t_2) \rightleftharpoons \overline{s_1, s_2} + 1 = \overline{t_1, t_2}$;
3. $plus(r_1, r_2, s_1, s_2, t_1, t_2) \rightleftharpoons \overline{r_1, r_2} + \overline{s_1, s_2} = \overline{t_1, t_2}$;
4. $times(r_1, r_2, s_1, s_2, t_1, t_2) \rightleftharpoons \overline{r_1, r_2} \times \overline{s_1, s_2} = \overline{t_1, t_2}$;
5. $equal(s_1, s_2, t_1, t_2) \rightleftharpoons \overline{s_1, s_2} = \overline{t_1, t_2}$;

Proof. The following program together with the clauses from Lemma 2.2 computes the relations:

```
zero(X,X).

suc(Y1,Y2,f(Y1,...,Y1),Y2) :-
   numeral(Y1,Y2).

plus(X1,X2,Y1,Y2,X1,X2) :-
   zero(Y1,Y2),
   numeral(X1,X2).
plus(X1,X2,Y1,Y2,Z1,Z2) :-
   suc(U1,U2,Y1,Y2),
   suc(V1,V2,Z1,Z2),
   plus(X1,X2,U1,U2,V1,V2).

times(X1,X2,Y1,Y2,Z1,Z2) :-
   zero(Y1,Y2),
   zero(Z1,Z2),
   numeral(X1,X2).
times(X1,X2,Y1,Y2,Z1,Z2) :-
   suc(U1,U2,Y1,Y2),
   times(X1,X2,U1,U2,V1,V2),
   plus(V1,V2,X1,X2,Z1,Z2).

equal(X1,X2,Y1,Y2) :-
   numeral(X1,X2),
   numeral(Y1,Y2),
   eq(X1,Y1),
   eq(X2,Y2).

eq(X,X).
```

To check it one can apply Lemma 2.1. ■

Theorem 2.2 *Let n be a natural number and R be any recursively enumerable relation on n-tuples of natural numbers. Then the relation $\Re$ on $Term^n$ defined by*

$$\begin{aligned}\Re(s_1, s_2, \ldots, t_1, t_2) \rightleftharpoons\ & \langle s_1, s_2\rangle, \ldots, \langle t_1, t_2\rangle \\ & \textit{are numerals and} \\ & R(\overline{s_1, s_2}, \ldots, \overline{t_1, t_2}) \qquad (*)\end{aligned}$$

is H-computable.

Proof. By the theorem proved by Matiyasevič [6] the relation R can be represented in the form

$$R(x_1,\ldots,x_n)\rightleftharpoons\exists y_1,\ldots,y_k \\ (g(x_1,\ldots,x_n,y_1,\ldots,y_k) = \\ h(x_1,\ldots,x_n,y_1,\ldots,y_k))$$

where g, h are polynomials (i.e. terms composed only of $0,', +, \times$. Let *zero, suc, plus, times* be the predicates representing these functions i.e.

$$zero(x) \equiv\ x = 0$$

$$suc(x,y) \equiv\ x' = y$$

$$plus(x,y,z) \equiv\ x + y = z$$

$$times(x,y,z) \equiv\ x \times y = z$$

Using the equivalences

$$\varphi(0) \equiv\ \exists v(zero(v)\&\varphi(v))$$

$$\varphi(s(x)) \equiv\ \exists v(suc(x,v)\&\varphi(v))$$

$$\varphi(x+y) \equiv\ \exists v(plus(x,y,v)\&\varphi(v))$$

$$\varphi(x\times y) \equiv\ \exists v(times(x,y,v)\&\varphi(v))$$

we can rewrite equation $*$ in the form

$$R(v_1,\ldots,v_n)\rightleftharpoons\exists v_{n+1},\ldots,v_m(\varphi(v_1,\ldots,v_m))$$

where φ is a conjunction of atomic formulas of one of the following forms: $zero(v_i)$, $suc(v_i,v_j)$, $plus(v_i,v_j,v_k)$, $times(v_i,v_j,v_k)$, $v_i = v_j$, where v_i, v_j, v_k are not necessarily distinct.

We define the translation $\varphi^\star$ of such conjunctions in the following way:

$$(\psi_1\&\psi_2)^\star = \psi_1^\star, \psi_2^\star$$

$$zero(v_i)^\star = \texttt{zero}(v_{2i}, v_{2i+1})$$

$$suc(v_i,v_j)^\star = \texttt{suc}(v_{2i}, v_{2i+1}, v_{2j}, v_{2j+1})$$

$$plus(v_i,v_j,v_k)^\star = \\ \texttt{plus}(v_{2i}, v_{2i+1}, v_{2j}, v_{2j+1}, v_{2k}, v_{2k+1})$$

$$times(v_i,v_j,v_k)^\star = \\ \texttt{times}(v_{2i}, v_{2i+1}, v_{2j}, v_{2j+1}, v_{2k}, v_{2k+1})$$

$$(v_i = v_j)^\star = \texttt{equal}(v_{2i}, v_{2i+1}, v_{2j}, v_{2j+1})$$

Now consider the logic program $\mathcal{P}$, which is obtained from the program from Lemma 2.3 by adding the clause

$\texttt{r}(v_1,\ldots,v_{2n})$ `:-` $\varphi^\star(v_1,\ldots,v_{2m})$.

By routine inspection of the construction and using Lemma 2.1 one can prove that `r` computes R in the program $\mathcal{P}$. ∎

Definition 2.3 *Let $\mathcal{P}$ be a Horn clause logic program and R be a relation on natural numbers. We say that a predicate* $\mathtt{r}(X_1, \ldots, X_{2k})$ *defined in $\mathcal{P}$* represents *R in $\mathcal{P}$ iff for every tuple $t_1, \ldots, t_{2k}$ of terms*

$\langle t_1, \ldots, t_{2k} \rangle \in S_{\mathcal{P},\mathtt{r}}$ *iff*
$\langle t_1, t_2 \rangle \ldots \langle t_{2k-1}, t_{2k} \rangle$ *are numerals and*
$R(\overline{t_1, t_2}, \ldots, \overline{t_{2k-1}, t_{2k}})$. ∎

We can reformulate Theorem 2.2 as follows: for every computable predicate R there exist a Horn clause program $\mathcal{P}$ and a predicate `r`, such that `r` represents R in $\mathcal{P}$.

Theorem 2.2 shows us a way to represent all computable functions on natural numbers by Horn clause logic programs having at least one function symbol in the language. The next step is to show that we can represent the enumeration γ. The technical problem is that we cannot distinguish variable numbers in the enumeration γ in programs, because all terms are equivalent up to renaming. The same situation somehow arises in metaprogramming, which is usually solved by encoding variables by terms as in [5]. In practical implementations of Prolog the problem is solved by introducing primitives like `var` or `==`. We will solve this technical problem by introducing "relativized statements"[3] in the subsequent lemmas.

2.3 Universe with a binary function

The proof of the main statement (Theorem 2.5 below) will be different depending on existence of function symbols of arity ≥ 2 in the set F. From now on we assume that `g` is a fixed function symbol from F with arity 2. (It is easy to change the proof if the arity of `g` is greater than 2, but we do not want to abuse the notations in this paper). The proof for the case when arity of symbols from F is restricted to 1, will be given below (Theorem 2.4). Let us also note that in general `g` may coincide with `f`.

Definition 2.4 *Let us introduce the notation "term t represents the list of natural numbers $[n_1, \ldots, n_k]$".*

1. *The term* `g(X,f(X,...,X))` *represents the empty list* $[]$;
2. *Let the term t represent the list $[n_1, \ldots, n_k]$, n_0 be a natural number, all the variables of the pair of terms s_1, s_2 be distinct from the variables of t, and $n_0 = \overline{s_1, s_2}$. Then the term $\mathtt{g}(\mathtt{g}(s_1, s_2), t)$ represents the list $[n_0, \ldots, n_k]$.* ∎

Lemma 2.4 *The relations*

1. *$R(s)$ — "The term s represents some list of numbers",*
2. *$M(t_1, t_2, s)$ — "t_1, t_2 is a numeral, s represents some list and the natural number $\overline{t_1, t_2}$ is a member of the list represented by s" and*

[3] Statements like "If n is the Gödel number of the variable v, then …

3. *$N(t_1, t_2, s)$ — "t_1, t_2 is a numeral, s represents some list and the natural number $\overline{t_1, t_2}$ is not a member of the list represented by s"*

are H-computable.

Proof. By Theorem 2.2 there is a Horn clause program $\mathcal{P}$, which represents the relation "m, n are non-equal natural numbers". Let this relation be computed by the predicate `nonequal(X1,X2,Y1,Y2)`. Consider the program, obtained from $\mathcal{P}$ by adding the definition of `numeral` from Lemma 2.2 and the following clauses:

```
list(g(X,f(X,...,X))).
list(g(g(Y1,Y2),L)) :-
   numeral(Y1,Y2),
   list(L).

member(Y1,Y2,g(g(Y1,Y2),L)) :-
   list(L),
   numeral(Y1,Y2).
member(Y1,Y2,g(g(Z1,Z2),L)) :-
   numeral(Z1,Z2),
   member(Y1,Y2,L)

omega(g(X,f(X,...,X))).

new(Y1,Y2,L) :-
   numeral(Y1,Y2),
   omega(L).
new(Y1,Y2,g(g(Z1,Z2),L)) :-
   nonequal(Y1,Y2,Z1,Z2),
   new(Y1,Y2,L).
```

Using properties of natural numbers and ω from lemma 2.1, one can prove that `list`, `member`, `new` satisfy the statement of the theorem. ■

Let *exponent* be any natural number of the form 2^n, *assignment* be any injective mapping from a finite subset V_0 of the set of variables V to the set of all exponents. Assignments will be denoted by $[x_1/n_1, \ldots, x_k/n_k]$. Whenever we use this notation, we will assume that all variables x_i are pairwise distinct and all numbers n_i are also pairwise distinct.

Definition 2.5 *Let us introduce the notation "the term t represents the assignment θ".*

1. *The term ω represents the empty assignment $[]$;*

2. *Suppose the term t represents the assignment $[x_1/n_1, \ldots, x_k/n_k]$, n_0 is an exponent, all the variables of the pair of terms s_1, s_2 are distinct from the variables of t, the variable x_0 does not occur in either of s_1, s_2, t, and $n_0 = \overline{s_1, s_2}$. Then the term $g(g(x_0, g(s_1, s_2)), t)$ represents the assignment $[x_0/n_0, x_1/n_1, \ldots, x_k/n_k]$.* ■

Lemma 2.5 *The relation $R(t)$ — "The term t represents some assignment" is H-computable.*

Proof. By Theorem 2.2 there is a Horn clause logic program $\mathcal{P}$, which represents the relation "n is an exponent". Let this relation be computed in $\mathcal{P}$ by the predicate `exponent(X1,X2)`. Consider the program, obtained from $\mathcal{P}$ by adding the following clauses:

```
assignment(A) :-
   assignment_aux(A,L).

assignment_aux(A,L) :-
   omega(A),
   omega(L).

assignment_aux(g(g(X,g(Y1,Y2)),A),
                 g(g(Y1,Y2),L)) :-
   assignment_aux(A,L),
   exponent(Y1,Y2),
   list(L),
   new(Y1,Y2,L).
```

To prove that this program computes the relation, it is sufficient to note that the predicate `assignment_aux` computes the following relation:

> `assignment_aux`$(t, s) \rightleftharpoons$ the term t represents an assignment $[x_1/n_1, \ldots, x_k/n_k]$, and the term s represents the list of numbers $[n_1, \ldots, n_k]$. ■

Let t be a term and all variables of t belong to the set $x_1, \ldots, x_k$. Let θ be an assignment $[x_1/n_1, \ldots, x_k/n_k]$. *The Gödel number $\gamma_\theta(t)$ of t under θ* is defined similar to the definition of Gödel numbering γ:

$$\gamma_\theta(x_i) = n_i$$

$$\gamma_\theta(f_i(t_1, \ldots, t_m)) = 3^i 5^{\gamma_\theta(t_1)} \times \ldots \times p_{m+1}^{\gamma_\theta(t_m)}$$

Lemma 2.6 *The following relation $R(t, l, s_1, s_2)$ on terms is H-computable:*

- *the term l represents an assignment $\theta = [x_1/n_1, \ldots, x_k/n_k]$ and*
- *all variables of the term t belong to the set $\{x_1, \ldots, x_k\}$ and*
- *The Gödel number $\gamma_\theta(t)$ of t under the assignment θ is equal to $\overline{s_1, s_2}$.*

Proof. Take the program consisting of the definitions of the relations `assignment_aux`, `new`, `member`, `equal`, `nonequal`, and `numeral` from Lemmas 2.2,2.3,2.4,2.5. Add to this program the following:

```
add_to(g(X,g(N1,N2)),A,A) :-
   exponent(N1,N2),
   assignment_aux(A,L),
   member(N1,N2,L),
   glue(g(X,g(N1,N2)),A).
```

```
add_to(g(X,g(N1,N2)),A,g(g(X,g(N1,N2)),A)) :-
   exponent(N1,N2),
   assignment_aux(A,L),
   new(N1,N2,L).

glue(g(X,g(N1,N2)),g(g(X,g(N1,N2)),A)) :-
   numeral(N1,N2).
glue(g(X,g(N1,N2)),g(g(Y,g(M1,M2)),A)) :-
   nonequal(N1,N2,M1,M2),
   glue(g(X,g(N1,N2)),A).
```

One can prove the following property of `add_to`: `add_to`$(g(t_1, g(s_1, s_2)), r_1, r_2)$ holds iff

1. t_1 is a variable.
2. r_1, r_2 represent some assignments θ_1, θ_2.
3. One of the following statements is true:
   (a) $t_1/\overline{s_1, s_2} \in \theta_1$ and θ_1 is identical to θ_2;
   (b) $t_1/\overline{s_1, s_2} \notin \theta_1$ and θ_2 is obtained from θ_1 by adding $t_1/\overline{s_1, s_2}$.

Now consider the following clauses:

```
adjoin(A1,A2,A2) :-
   assignment(A2),
   omega(A1).
adjoin(g(g(X,g(N1,N2)),A1),A2,A3) :-
   add_to(g(X,g(N1,N2)),A2,A4),
   adjoin(A1,A4,A3).
```

One can prove the following property of `adjoin`: `adjoin`(t_1, t_2, t_3) is true iff

1. t_1, t_2, t_3 represent assignments $\theta_1, \theta_2, \theta_3$.
2. For a variable v and a natural number n, $v/n \in \theta_3$ iff $v/n \in \theta_1$ or $v/n \in \theta_2$.

Let `f1`,...,`fn` be all function symbols from F. For each `fi` let $h_i(x_1, \ldots, x_m)$ be the function on natural numbers defined as

$$h(x_1, \ldots, x_m) = 3^i 5^{x_1} \times \ldots \times p_{m+1}^{x_m}$$

(cf. the definition of γ). Let $\mathcal{P}$ be a program such that the predicates `hi(X1,...,X2m)` represent h_i in $\mathcal{P}$. Consider the following clause:

```
gamma(X,L,Y1,Y2) :-
   exponent(Y1,Y2),
   assignment(L1)
   add_to(g(X,g(Y1,Y2)),L1,L).
```

and the clauses

```
gamma(f1(X1,...,Xk),L,Y1,Y2) :-
   gamma(X1,L1,Z1,V1),
   ⋮
   gamma(Xk,Lk,Zk,Vk),
   adjoin(L1,L2,M3),
   adjoin(L3,M3,M4),
   ⋮
   adjoin(Lk,Mk,L),
   hi(Z1,V1,...,Zk,Vk,Y1,Y2).
```

for each `fi` from F.

One can prove the following about `gamma`: `gamma`(s, t, r_1, r_2) holds iff

1. the term t represents an assignment $\theta = [x_1/n_1, \ldots, x_k/n_k]$.
2. the term s is a term whose all variables belong to the set $\{x_1, \ldots, x_k\}$.
3. $\gamma_\theta(s) = \overline{r_1, r_2}$. ∎

From this lemma we can prove

Theorem 2.3 *Let there be at least one symbol with arity ≥ 2 in F. Then every computable well-formed set $S \in Term^n$ is H-computable.*

Proof. Let R be the following set of natural numbers:

$$\{\langle\gamma(s_1), \ldots, \gamma(s_n)\rangle \mid \langle s_1, \ldots, s_n\rangle \in S\}$$

Since R is recursively enumerable, by Theorem 2.2 there exists a Horn clause logic program $\mathcal{P}$ and a predicate `r`, such that `r` represents R in $\mathcal{P}$. Add to the program $\mathcal{P}$ the definition of `gamma` from Lemma 2.6 and the following clause:

```
s(X1,...,Xn) :-
   r(Y1,Z1,...,Yn,Zn)
   gamma(X1,L,Y1,Z1),
   ⋮
   gamma(Xn,L,Yn,Zn).
```

Then

`s`$(t_1, \ldots, t_n) \leftrightarrow$
there are $y_1, z_1, \ldots, y_n, z_n$ and an assignment l, such that l contains all free variables of $t_1, \ldots, t_n$ and for every $i \in \{1, \ldots, n\}$ we have $\gamma_l(t_i) = \overline{y_i, z_i}$ and `r`$(y_1, z_1, \ldots, y_n, z_n) \leftrightarrow$
there are $y_1, z_1, \ldots, y_n, z_n$ and an assignment l, such that l contains all free variables of $t_1, \ldots, t_n$ and for every $i \in \{1, \ldots, n\}$ we have $\gamma_l(t_i) = \overline{y_i, z_i}$ and $\langle\overline{y_1, z_1}, \ldots, \overline{y_n, z_n}\rangle \in R \leftrightarrow$
there is an assignment l, such that l contains all free variables of $t_1, \ldots, t_n$ and $\langle\gamma_l(t_1), \ldots, \gamma_l(t_n)\rangle \in R \leftrightarrow$
$\langle t_1, \ldots, t_n\rangle \in S$. ∎

2.4 Universe with unary functions

In proving Theorem 2.3 we assumed that there is at least one symbol $g \in F$ of arity ≥ 2. This function symbol g had been used to encode assignments. To represent assignments we needed lists of an arbitrary length, because the number of distinct variables in $Term^n$ is not restricted. Assume now that all function symbols from F have arity 0 or 1. The following lemma gives us a hint how to handle this case.

Lemma 2.7 *Let all symbols from F have arity 0 or 1. Then any tuple $\tau \in Term^n$ has at most n occurrences of variables.* ■

The following lemma is an analog of Lemma 2.6.

Lemma 2.8 *The following relation $R(x_1, k_1, m_1, \ldots, x_n, k_n, m_n, t, k, m)$ is H-computable:*

> $\langle k_1, m_1\rangle, \ldots, \langle k_n, m_n\rangle$ *are numerals and*
> $x_1, \ldots, x_n$ *are distinct variables and*
> $\theta = [x_1/\overline{k_1, m_1}, \ldots, x_n/\overline{k_n, m_n}]$ *is a substitution and*
> $\langle k, m\rangle$ *is a numeral and*
> $\gamma_\theta(t) = \overline{k, m}$.

Proof. Let `fi`, $i \in \{1, \ldots, l\}$, be all unary function symbols and `fj`, $j \in \{l+1, \ldots, q\}$, be all constants from F. For each `fi` let $h_i(x)$ be the functions on natural numbers defined as

$$h_i(x) = \begin{cases} 3^i 5^x, & \text{if } i \leq l \\ 3^i, & \text{if } i > l \end{cases}$$

(cf. the definition of γ). Let $\mathcal{P}$ be a program such that the predicates `hi(X1,...,X2m)` represent h_i in $\mathcal{P}$. Consider the program obtained from $\mathcal{P}$ by adding the following rules.

1. n rules of the form

   ```
   r(X1,M1,K1,...,Xn,Mn,Kn,Xi,Ki,Mi) :-
      exponent(M1,K1),
      ⋮
      exponent(Mn,Kn),
      nonequal(M1,K1,M2,K2),
      nonequal(M1,K1,M3,K3),
      ⋮,
      nonequal(Mn-1,Kn-1,Mn,Kn).
   ```

   for $i \in \{1, \ldots, n\}$.

2. l rules of the form

   ```
   r(X1,M1,K1,...,Xn,Mn,Kn,fi(T),K,M) :-
      r(X1,M1,K1,...,Xn,Mn,Kn,T,K0,M0),
      hi(k0,m0,K,M).
   ```

for $i \in \{1, \ldots, l\}$.

3. $q - l$ rules of the form

```
r(X1,M1,K1,...,Xn,Mn,Kn,fi,K,M) :-
   r(X1,M1,K1,...,Xn,Mn,Kn,T,K0,M0),
   hi(K0,M0,K,M).
```

for $i \in \{l+1, \ldots, q\}$.

The proof that `r` computes R is similar to that of Lemma 2.6. ■

Theorem 2.4 *Let all function symbols from F have arity 0 or 1. Then any computable well-formed set $S \in Term^n$ is H-computable.*

Proof. Similar to that of Theorem 2.3, but based on Lemmas 2.7 and 2.8. ■

2.5 Main theorem

Combining theorems 2.1,2.3 and 2.4, we get

Theorem 2.5 *A well-formed set $S \in Term^n$ is computable iff it is H-computable.*

Proof. In one direction follows from theorems 2.1,2.3 and 2.4. In the other direction by standard arguments of the theory of recursive functions [8]. ■

It is possible to generalize Theorem 2.5 also to the case of C-semantics. The additions to the above proofs are minor. The major change is the definition of well-formed sets: they should be closed under substitution instead of renaming. All the basic ideas remain the same.

3 Applications

Here we show an application of our results to the problem of constructing *metainterpreters*. One of the interesting questions in comparing two logic programming languages is the existence of a metainterpreter for one logic programming language written in another logic programming language (for a more complete discussion on the subject see e.g. [11]). Metainterpreter for a language may simulate only some aspects of the language's observable behavior. Here we shall show that for S-semantics there is a metainterpreter for a large subset of Prolog written in the language of pure Horn clauses.

Let *logic program* $\mathcal{P}$ be any finite list $R_1, \ldots, R_k$ of *facts* of the form A and *rules* of the form $A : -A_1, \ldots, A_n$. Assume that the language includes the constant `[]` and the binary list constructor `[|]` (we shall use Prolog notations for lists). Let the image $im(A)$ of such a fact be the list `[`A`]` and the image $im(A : -A_1, \ldots, A_n)$ of such a rule be the list `[`$A, A_1, \ldots, A_n$`]`. We say that the list `[`$im(R_1), \ldots, im(R_n)$`]` *represents* the logic program $\mathcal{P}$ iff for every $i \neq j$ all the variables of $im(R_i)$ are distinct from the variables of $im(R_j)$.

Let us now informally consider how Prolog works. We do not specify now what variant of the real Prolog we have in mind. Given a query q with variables $v_1, \ldots, v_n$

and a program $\mathcal{P}_1$ such that the variables $v_1, \ldots, v_n$ do not occur in $\mathcal{P}_1$, it gives as the result of computations a substitution $v_1 = t_1, \ldots, v_n = t_n$ or answers "no". The answer substitutions may contain variables, these variables must be different from $v_1, \ldots, v_n$ and from all the variables of the program $\mathcal{P}_1$. Let $\mathcal{P}$ be a Horn clause logic program written in a language $\mathcal{L}$. It is called *metainterpreter*[4] for Prolog iff it defines a ternary predicate `call`, such that $\mathtt{call}(t_1, t_2, t_3) \in S_{\mathcal{P},P}$ iff

1. t_3 represents a Prolog program $\mathcal{P}_1$,
2. t_1 is a query to $\mathcal{P}_1$ and
3. t_2 is the answer substitution, computed by Prolog.

Now we restrict ourselves to e.g. standard Prolog with the leftmost choice rule, depth-first computation rule and primitives `!`, `not`, `var`, `==`.

Theorem 3.1 *There exist a metainterpreter for Prolog written in the language of pure Horn clauses.*

Proof. Trivially follows from Theorem 2.5. It is sufficient to note that the set

$$\{\mathtt{call}(t_1, t_2, t_3) \mid t_3 \text{ represents a Prolog program } \mathcal{P}_1,\ t_1 \text{ is a query to } \mathcal{P}_1 \text{ and } t_2 \text{ is an answer substitution computed by Prolog}\}$$

is well-formed, i.e. closed under substitution. ∎

This theorem shows a fundamental difference between our results and the results from [12, 1, 9]. The standard results are not applicable to prove Theorem 3.1. The presence of variables requires completely different encodings.

It is possible to generalize Theorem 3.1 to much richer variants of Prolog. The features that may violate the property to be well-formed are input-output relations and relations expanding the Herbrand universe. Our theorem shows that such strong primitives of Prolog as `var`, `!` and standard Prolog's choice rule and computation rule do not extend expressive power of Horn clauses, at least at the level of S-semantics. As for C-semantics, the theorem is not true, because the real Prolog is not monotonic. For example, the query `var(t)` gives an answer substitution when `t` is a variable, but fails when `t` is any other term. Thus, we can not capture semantics of real Prolog by C-semantics.

There is another immediate application of our results which also shows that ordinary semantics do not capture the semantics of Prolog from the computational viewpoint:

Theorem 3.2 *Let L be a language having at least one function symbol. Then there is a Horn clause logic program $\mathcal{P}$, which defines a predicate A, with the following properties:*

[4] S-semantics reflects only a small part of the behavior of a program. So in this case we might use "metaevaluator" instead of "metainterpreter". However the following theorem 3.1 can be proved for any reasonable definition of a metainterpreter.

1. *Each ground atom of the form $A(t)$ in the language L is provable from $\mathcal{P}$.*

2. *The set of all non-ground atoms of the form $A(t)$ provable from $\mathcal{P}$ is undecidable.*

Proof. Consider the case when L contains a unary function symbol f. Let R be any undecidable set. Take the following set of atomic formulas

$$S = \{t \mid t \text{ is ground or takes the form } f^n(X),\ n \in R\}$$

and apply Theorem 2.5. ■

References

[1] H.Andréka and I.Németi. The generalized completeness of horn clause logic seen as a programming language. *Acta Cybernetica*, 4:3–10, 1978.

[2] Yu.L.Ershov. *Theory of enumerations (in Russian).* Nauka, 1977.

[3] M.Falaschi, G.Levi, M.Martelli, and C.Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):298–318, 1989.

[4] L.Henshen and L.Wos. Unit refutation and Horn sets. *Journal of the Association for Computing Machinery*, 21:590–605, 1974.

[5] P.M.Hill and J.W.Lloyd. The Gödel report. Technical Report TR-91-02, Univ. of Bristol, Department of Computer Science, September 1991.

[6] Matiyasevič. The diophantiness of recursively enumerable sets (in Russian). *Soviet Mathematical Doklady*, pages 279–282, 1970.

[7] M.Minsky. *Computation: finite and infinite machines.* Prentice Hall, 1967.

[8] H.Rogers. *Theory of recursive functions and effective computability.* McGraw Hill, 1967.

[9] P. Štěpánek and J.Šebelík. Horn clause programs for recursive functions. In K.L.Clark and S.-Å.Tärnlund, editors, *Logic Programming*, pages 325–340. Academic Press, 1982.

[10] J.C.Sheperdson. Negation in logic programming. In J.Minker, editor, *Deductive Databases and Logic Programming*, pages 19–88. Morgan Kaufmann, 1988.

[11] E.Shapiro. A family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[12] S.-Å. Tärnlund. Horn clause computability. *BIT*, 17:215–216, 1977.

[13] A. Voronkov. On computability by logic programs. In H. Blair, W. Marek, A. Nerode, and J. Remmel, editors, *Proceedings of the Workshop "Structural Complexity and Recursion-Theoretic Methods in Logic Programming"*, pages 165–174, Washington, D.C., November 1992.