INFORMATION AND CONTROL 56, 212–219 (1983)

A Necessary and Sufficient Condition in Order That a Herbrand Interpretation Be Expressive Relative to Recursive Programs

PAWEŁ URZYCZYN

Institute of Mathematics, University of Warsaw, 00-901 Warszawa, PKiN, Poland

It is proved that a recursive program (without counters) is able to enumerate all elements in any Herbrand interpretation. It follows that all recursive program domains in a Herbrand interpretation can be defined by first-order formulas iff there are first-order formulas expressing integer arithmetic in that interpretation.

1. INTRODUCTION

We say that an interpretation $\mathbf{A} = (A, =, r_1^{\mathbf{A}},..., r_n^{\mathbf{A}}, f_1^{\mathbf{A}},..., f_m^{\mathbf{A}})$ is *expressive* relative to a class of programs $\mathscr{P}$ iff the domain of each program $P \in \mathscr{P}$ in the similarity type of $\mathbf{A}$ can be defined in $\mathbf{A}$ by a first-order formula. That is, there exists a first-order formula φ such that, for all inputs $\bar{a}$ in $\mathbf{A}$,

$$P \text{ converges in } \mathbf{A} \text{ on the input } \bar{a} \text{ iff } \mathbf{A}, \bar{a} \models \varphi.$$

The notion of expressiveness plays an important role in the theory of Hoare logics (see, e.g., Cook, 1978). The present paper was motivated by the work of Clarke, German, and Halpern (1982), where the existence of Hoare logics for recursive programs was discussed, with the restriction to Herbrand interpretations only. (A *Herbrand interpretation* is an arbitrary interpretation generated by its underlying constants, i.e., zero-ary operations.) In particular, they proved that, for any Herbrand interpretation $\mathbf{A}$, if $\mathbf{A}$ is expressive relative to recursive programs then one of the following cases hold:

There are exists a bijection $F: A \to \omega$, and first-order formulas $Zero(x)$, $Succ(x, y)$, $Add(x, y, z)$, $Mult(x, y, z)$, in the similarity type of $\mathbf{A}$, such that, for all $a, b, c \in \mathbf{A}$,

$$\begin{aligned}
\mathbf{A}, a &\models Zero(x) &&\Leftrightarrow F(a) = 0,\\
\mathbf{A}, a, b &\models Succ(x, y) &&\Leftrightarrow F(a) + 1 = F(b),\\
\mathbf{A}, a, b, c &\models Add(x, y, z) &&\Leftrightarrow F(a) + F(b) = F(c),\\
\mathbf{A}, a, b, c &\models Mult(x, y, z) &&\Leftrightarrow F(a) \cdot F(b) = F(c).
\end{aligned} \qquad (1.1)$$

For each recursive program P, there exists an $n \in \omega$, such that each converging computation of P, on any input in $\mathbf{A}$, takes at most n steps. (1.2)

Interpretations satisfying condition (1.1) are called *strongly arithmetical.*

In the present paper we strengthen the above result, namely, we prove that a Herbrand interpretation is expressive relative to recursive programs iff it is strongly arithmetical or finite (Theorem 2.3). The main tool used in the proof is Theorem 2.2, which states that a recursive program is able to enumerate all elements in an arbitrary Herbrand interpretation (over a fixed similarity type).

It is worth noting that another proof of our main result can be obtained using Lemma 2.1, which is weaker than Theorem 2.2. Namely, Lemma 2.1 suffices to prove that any Herbrand interpretation satisfying (1.2) is finite (see Urzyczyn, 1983). Such a proof, however, would depend upon the complicated proof of the mentioned result of Clarke *et al.* Another reason for choosing a direct proof is that Theorem 2.2 seems to be of independent interest.

Before proving the results, we must explain the notion of a recursive program (we make no distinction between the notions: "program" and "program schema"). Informally, a *recursive program* is a finite flow-chart schema S, which may use names of programs including S itself as function (or relation) symbols, together with all programs whose names occur in S. A computation of a recursive program may thus include a sequence of recursive calls. We assume that no additional equipment (especially counters) is allowed in recursive programs. A precise definition should be formulated in the spirit of Constable and Gries (1972), with the additional assumption that programs can always test for equality. Nevertheless, we will use graphical descriptions and informal explanations in the proofs, instead of complicated ALGOL-like expressions. Although, in contrast to Clarke *et al.* (1982), we have chosen one particular class of programs, it should be clear that our consideration applies for many other classes, not less powerful than our recursive programs. In particular, Theorems 2.2 and 2.3 remain true for any "acceptable programming language with recursion" in the sense of the latter paper.

2. Main Results

Let σ be a fixed similarity type, i.e., a finite sequence of relation and function symbols: $=, r_1, r_2, \ldots, r_n, f_1, \ldots, f_m$, each equipped with a nonnegative arity. Below we assume that all programs and formulas under consideration, if not defined otherwise, are of similarity type σ, i.e., no other function or relation symbol may occur in them (except names of recursive procedures in programs). Also, all interpretations are assumed to be of similarity type σ, i.e., of the form $\mathbf{A} = (A, =, r_1^{\mathbf{A}}, r_2^{\mathbf{A}},..., r_n^{\mathbf{A}}, f_1^{\mathbf{A}},..., f_m^{\mathbf{A}})$, where $=$ is assumed to be interpreted as equality. Now, if S is a program with k input variables, then S defines a k-ary partial function (or relation) in any interpretation $\mathbf{A}$. The function (relation) obtained in that way is denoted by $S^{\mathbf{A}}$.

For simplicity in the definition to follow, and with no loss of generality, we assume at the moment that all the function symbols $f_1, f_2,..., f_m$ have the same arity $p \geqslant 1$. Consider an arbitrary interpretation $\mathbf{A}$, as above. For an arbitrary $\bar{a} \in A^k$, where k is a positive integer, we introduce the notion of the *natural chain* from $\bar{a}$, denoted $\{q_i: i < \beta\}$, for some ordinal $\beta < \omega + 1$, which is an appropriately defined linear ordering of a subset of the substructure generated in $\mathbf{A}$ by $\bar{a}$. The definition goes by induction:

(i) If all the elements in $\{a_1,..., a_k\}$, where $\bar{a} = (a_1,..., a_k)$, are distinct then the first k elements in the chain, $q_0,..., q_{k-1}$, are $a_1,..., a_k$, respectively. If only $k' < k$ elements in $\{a_1,..., a_k\}$ are distinct, then the first k' elements in the chain are the k' distinct entries of $\bar{a}$, listed in order of increasing index.

(ii) Suppose that the first l elements $\{q_i: i < l\}$ have been defined, for some $l < \omega$. To define the next element q_l we proceed as follows: We list in lexicographic order all sequences of the form $(i_1,..., i_p, j)$, where $i_1,..., i_p < l$ and $j \in \{1, 2,..., m\}$, and we choose the first such sequence for which $f_j^{\mathbf{A}}(q_{i_1},..., q_{i_p}) \notin \{q_i: i < l\}$. If such a sequence exists, we set $q_l = f_j^{\mathbf{A}}(q_{i_1},..., q_{i_p})$, otherwise q_l is undefined and the natural chain is finite.

We leave it to the reader to adapt the above definition to the case when not all of the function symbols in σ have the same arity $p \geqslant 1$ (which also includes the possibility of zero-ary, i.e., constant symbols). The proof of Lemma 2.0 is straightforward and is left to the reader.

LEMMA 2.0. *Let* $\mathbf{A}$ *be an arbitrary interpretation and* $\bar{a} \in A^k$.

(1) *If* $\bar{a}$ *generates finitely many elements then the natural chain from* $\bar{a}$ *includes all the elements generated by* $\bar{a}$.

(2) *If* $\bar{a}$ *generates infinitely many elements then the natural chain from* $\bar{a}$ *is infinite, but does not necessarily include all the elements generated by* $\bar{a}$.

LEMMA 2.1. *Let* $k \geqslant 1$. *There exists a recursive program* $\mathrm{NEXT}(x_1,..., x_k, y)$ *such that, for every interpretation* $\mathbf{A}$ *and every* $\bar{a} \in A^k$, $b \in A$, *if* $\{q_i: i < \beta\}$ *is the natural chain from* $\bar{a}$, *then*

$$\begin{aligned} \mathrm{NEXT}^{\mathbf{A}}(\bar{a}, b) &= q_{j+1}, && \text{if } b = q_j, \text{for some } j \text{ with } j+1 < \beta, \\ &= b, && \text{if } b = q_j \text{ and } j+1 = \beta, \\ &= \text{undefined}, && \text{if } b \neq q_j, \text{for all } j < \beta. \end{aligned}$$

Proof. As before we assume that all function symbols in σ are of the same arity $p \geqslant 1$. For a given interpretation $\mathbf{A}$, denote by *next* the partial function to be computed by the program NEXT in $\mathbf{A}$. Clearly, the element q_i in the natural chain from $\bar{a}$ is none other than

$$next(\bar{a}, next(\bar{a}, ..., next(\bar{a}, next(a, a_1)) ...)).$$

On an input $\bar{a}$, b, the program NEXT operates as follows: If $b = a_i$, for some $i < k$, it computes the value $next(\bar{a}, b)$ by executing a sequence of conditional instructions. Otherwise, by calling itself repeatedly, NEXT can generate the successive elements in the natural chain from $\bar{a}$, until it reaches b. (At this moment, we can assume that $b = q_j$, for some j, and that $\text{NEXT}^{\mathbf{A}}(\bar{a}, q_i) = next(\bar{a}, q_i) = q_{i+1}$ holds for all $i < j$.) In this case, the program searches for the first (lexicographically) sequence $(i_1, ..., i_p, l)$, such that $i_1, ..., i_p < j$ and $f_l^{\mathbf{A}}(q_{i_1}, ..., q_{i_p}) \notin \{q_0, q_1, ..., q_j\}$, and produces the output $f_l^{\mathbf{A}}(q_{i_1}, ..., q_{i_p})$. If there is no such sequence, NEXT stops with the output equal to b.

The details of the program NEXT are given on Fig. 1. Its verification is left to the reader.

As we observed above, it may happen that some of the elements generated by $\bar{a}$ do not occur in the natural chain from $\bar{a}$. One can, however, improve our program NEXT so that it produces a chain of all elements generated by any input.

THEOREM 2.2. *For every* $k \geqslant 1$, *there is a program* $\text{SUC}(x_1, ..., x_k, y)$ *such that, if* $\mathbf{A}$ *is an arbitrary interpretation,* $\bar{a} \in \mathbf{A}^k$ *and* $code\colon \omega \to \mathbf{A}$ *is a function defined by*

$$code(0) = a_1,$$

$$code(n+1) = \text{SUC}^{\mathbf{A}}(\bar{a}, code(n)),$$

then the set $\{code(n)\colon n \in \omega\}$ *coincides with the substructure generated by* $\bar{a}$ *in* $\mathbf{A}$.

Proof. For the proof we introduce the notion of a *recursive program with counters*, which is a recursive program in the similarity type σ extended by the arithmetical symbols 0 and s, for zero and successor, respectively. There are special variables, called counters, that range over the set ω of nonnegative integers and occur in the instructions containing arithmetical symbols. It is easy to define a recursive program with counters, CSUC, satisfying the statement of the theorem with CSUC replacing SUC throughout. (The existence of CSUC follows also from the results of Section 10 in Constable and Gries (1972).

First we define a recursive program SUC_1 without counters to simulate CSUC on inputs defining infinite subalgebras. For an input $(\bar{a}, b)$, SUC_1 uses

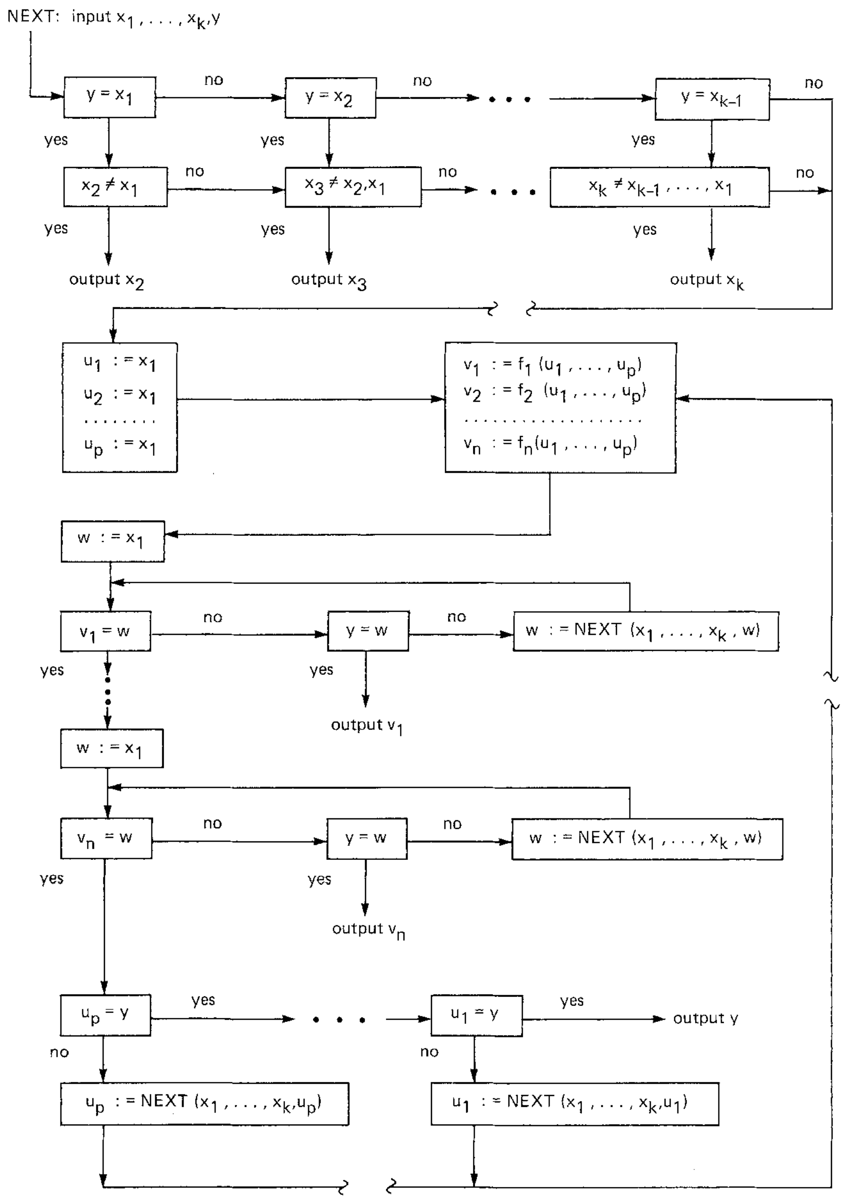
NEXT: input $x_1, \ldots, x_k, y$
$y = x_1$
$y = x_2$
$y = x_{k-1}$
$x_2 \neq x_1$
$x_3 \neq x_2, x_1$
$x_k \neq x_{k-1}, \ldots, x_1$
yes
no
output x_2
output x_3
output x_k
$u_1 := x_1$
$u_2 := x_1$
$u_p := x_1$
$v_1 := f_1(u_1, \ldots, u_p)$
$v_2 := f_2(u_1, \ldots, u_p)$
$v_n := f_n(u_1, \ldots, u_p)$
$w := x_1$
$v_1 = w$
$y = w$
$w := \text{NEXT}(x_1, \ldots, x_k, w)$
output v_1
$v_n = w$
output v_n
$u_p = y$
$u_1 = y$
output y
$u_p := \text{NEXT}(x_1, \ldots, x_k, u_p)$
$u_1 := \text{NEXT}(x_1, \ldots, x_k, u_1)$

FIGURE 1

variables ranging over the natural chain from $\bar{a}$, instead of counters. The successor operation is simulated by NEXT, and a_1 plays the role of zero. Whenever NEXT is applied to a vector $(\bar{a}, c)$, the main program checks whether its output differs from c or not. If not, SUC_1 jumps to a special exit labelled "abort" and gives no output. Observe that c must occur in the natural chain from $\bar{a}$, hence NEXT must converge on $(\bar{a}, c)$.

Suppose for a moment that $\mathrm{NEXT}^{\mathbf{A}}(\bar{a}, c) \neq c$ in all cases occurring in the computation. This means that the simulation is successful, i.e., either $\mathrm{SUC}_1^{\mathbf{A}}(\bar{a}, b) = \mathrm{CSUC}^{\mathbf{A}}(\bar{a}, b)$, or both are undefined. A difficulty appears when the simulation fails; that is, when $\mathrm{NEXT}^{\mathbf{A}}(\bar{a}, c) = c$ for some c used as a value of a "counter." In the latter case, $\bar{a}$ can only generate a finite number of elements, and NEXT may be used to compute all of them, but not necessarily in the order given by CSUC. The latter remark means that the algorithm of SUC_1 cannot be completed by simply producing the output $\mathrm{NEXT}^{\mathbf{A}}(\bar{a}, b)$ in this case.

Our program SUC must be more complicated. In order to define it we first construct a program SUC_2, which works as follows: on input $(\bar{a}, b)$, it sets $z := x_1$ and, while $z \neq b$, it iterates the procedure $z := \mathrm{SUC}_1(\bar{x}, z)$; at last it executes once more $z := \mathrm{SUC}_1(\bar{x}, z)$ and sets the value of z to the output. Of course, if SUC_2 gives an output then the output is equal to that produced by SUC_1. In particular, it works successfully, provided $\bar{a}$ generates infinitely many elements. An important property of SUC_2 is that if $\mathrm{SUC}_2^{\mathbf{A}}(\bar{a}, b)$ is defined, then $b = code(n)$, for some n, and $\mathrm{SUC}_2^{\mathbf{A}}(\bar{a}, code(i)) = code(i+1)$ holds for all $i < n$.

The program SUC executes the following algorithm: For any given input $(\bar{a}, b)$ it first executes SUC_2 on that input. If SUC_2 gives an output, this is also the output of SUC. Otherwise (when SUC_2 reaches an abort-labelled exit), the program searches for the first element c in the natural chain from $\bar{a}$ that cannot be obtained from $\bar{x} = \bar{a}$, $z = a_1$, by iterating $z := \mathrm{SUC}_2(\bar{x}, z)$. For this, elements of the natural chain are successively enumerated, and each time a new element is produced, all the iterations of SUC_2 are repeated. The value c is set to the output. It may happen, however, that such a c does not exist. Then SUC outputs b, since b must be the last element in the sequence *code*. ▮

Theorem 2.2 enables us to prove our necessary and sufficient condition for the expressiveness of Herbrand interpretations.

THEOREM 2.3. *Let* **A** *be a Herbrand interpretation. Then the following conditions are equivalent*:

(i) **A** *is expressive relative to recursive programs.*

(ii) **A** *is strongly arithmetical or finite.*

Proof. Assume that **A** is infinite and (i) holds. With no loss of generality, assume there is only one zero-ary function symbol f in σ. Let SUC be a program satisfying the property described in Theorem 2.2, for $k = 1$. Let *code*: $\omega \to A$ be defined by

$$code(0) = f^{\mathbf{A}},$$
$$code(n+1) = \mathrm{SUC}^{\mathbf{A}}(f^{\mathbf{A}}, code(n)).$$

Further, let $F = code^{-1}: A \to \omega$. It is straightforward that F is a bijection and that, for any $a, b \in A$,

$$\mathrm{SUC}^{\mathbf{A}}(f^{\mathbf{A}}, a) = b \qquad \text{iff} \quad F(a) + 1 = F(b).$$

Consider the program SUCC(x, y), which converges iff SUC(f, x) converges and gives output equal to y. Clearly, any formula defining the domain of SUCC$^{\mathbf{A}}$ can be chosen as *Succ*(x, y) in (1.1). It is left to the reader to check that there are recursive programs ADD(x, y, z) and MULT(x, y, z), satisfying

$$\mathrm{ADD}^{\mathbf{A}}(a, b, c) \text{ converges} \Leftrightarrow F(a) + F(b) = F(c),$$
$$\mathrm{MULT}^{\mathbf{A}}(a, b, c) \text{ converges} \Leftrightarrow F(a) \cdot F(b) = F(c),$$

for all a, b, $c \in A$. The formulas *Add* and *Mult* are thus defined as arbitrary formulas defining the domains of the above programs. Of course, *Zero*(x) is the formula "$x = f$." We conclude that **A** is strongly arithmetical.

It is easily seen that every finite Herbrand interpretation is expressive relative to any class of programs. Thus, suppose that **A** is a strongly arithmetical Herbrand interpretation. We may assume that the domain of **A** is the set ω. Thus, its underlying relations $r_1^{\mathbf{A}}, \dots, r_n^{\mathbf{A}}$ and operations $f_1^{\mathbf{A}}, \dots, f_m^{\mathbf{A}}$ can be seen as relations and operations in ω. Let $\langle\ \rangle$ denote a recursive pairing function and let

$$\begin{aligned} R = \{\langle nr, \langle in, out\rangle\rangle &: in = \langle in_1 \langle in_2 \langle \cdots \langle in_{p-1}, in_p\rangle \cdots\rangle\rangle \text{ and} \\ &(1 \leqslant nr \leqslant n \wedge r_{nr}^{\mathbf{A}}(in_1, \dots, in_p) = \text{true} \wedge \mathbf{out} = 1) \\ &\vee (1 \leqslant nr \leqslant n \wedge r_{nr}^{\mathbf{A}}(in_1, \dots, in_p) = \text{false} \wedge \mathbf{out} = 0) \\ &\vee (n + 1 \leqslant nr \leqslant n + m \wedge f_{nr-n}^{\mathbf{A}}(in_1, \dots, in_p) = \mathbf{out})\}. \end{aligned}$$

It is easily verified that each program domain is recursively enumerable in R. By Rogers (1967, Theorem 7, Chap. 14), any such domain is definable by a first-order formula in the similarity type $(0, s, +, \cdot)$ extended by a predicate symbol **R** for the set R. Since the interpretation **A** is strongly arithmetical, we may express the arithmetical operations by formulas in the similarity type

σ, with help of *Zero*, *Add*, *Mult*, and *Succ*. The predicate symbol $\mathbf{R}(x)$ can be replaced by a formula using arithmetical symbols and the symbols in σ. Replacing again the arithmetical predicates by *Zero*, *Add*, *Mult*, and *Succ*, we get a formula in the similarity type σ, expressing the domain of a given recursive program. ∎

ACKNOWLEDGMENTS

I wish to thank Jerzy Tiuryn, for pointing out that Theorem 2.3, can be used for characterizing expressiveness. I also wish to thank A. J. Kfoury, for his critical remarks.

REFERENCES

CLARKE, E. M., GERMAN, S. M., AND HALPERN, J. Y. (1982), "Effective Axiomatizations of Hoare Logics," Research report, Harvard Univ., Cambridge.

CONSTABLE, R. L., AND GRIES, D. (1972), On classes of program schemata, *SIAM J. Comput.* **1** (1), 66–118.

COOK, S. A. (1978), Soundness and completeness of an axiom system for program verification, *SIAM J. Comput.* **7** (1), 70–90.

ROGERS, H. JR. (1967), "Theory of Recursive Functions and Effective Computability," McGraw–Hill, New York.

URZYCZYN, P. (1983), Non-trivial definability by flow-chart programs, *Inform. and Control*, in press.