INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Computability of String Functions Over Algebraic Structures

(Preliminary Version)

Armin Hemmerling*

TR-96-028

August 1996

Abstract

We present a model of computation for string functions over single–sorted, total algebraic structures and study some features of a general theory of computability within this framework. Our concept generalizes the Blum–Shub–Smale setting of computability over the reals and other rings. By dealing with strings of arbitrary length instead of tuples of fixed length, some suppositions of deeper results within former approaches to generalized recursion theory become superfluous. Moreover, this gives the basis for introducing computational complexity in a BSS–like manner. Relationships both to classical computability and to Friedman's concept of eds computability are established. Two kinds of nondeterminism as well as several variants of recognizability are investigated with respect to interdependencies on each other and on properties of the underlying structures. For structures of finite signatures, there are universal programs with the usual characteristics. In the general case (of not necessarily finite signature), the existence of universal functions is equivalent to the effective encodability of the structures, whereas the existence of m–complete sets turns out to be independent on those properties.

*Ernst Moritz Arndt University Greifswald, Department of Mathematics and Computer Science F.-L.-Jahn Str. 15a, D–17487 Greifswald, Germany; e-mail: hemmerli@rz.uni-greifswald.de
Parts of this work were done while the author was visiting the ICSI.

1 Introduction

In this paper, we present a handy model of computation over algebraic structures and demonstrate its application by dealing with some features of a general recursion theory. Our notion generalizes the concept of computability over the field of real numbers and other rings, as it has been introduced and, in particular with respect to time complexity, successfully applied by L. Blum, M. Shub and S. Smale (BSS) in their seminal paper [8]. For further presentations and surveys of BSS theory, see [5, 6, 44, 45, 7].

More precisely, we consider sequential computability of (partial) string functions and recognizability of sets of strings over the universe of a given structure. The latter is also allowed to be of infinite signature, but we restrict ourselves to single-sorted structures with total base functions and relations. The model enables us to deal with two fundamental types of nondeterminism. Moreover, it gives an appropriate basis for dealing with computational complexity in a BSS–like setting. This, however, will not be a subject of the present paper; for a previous information, the reader is referred to [30].

Since the thirties, when the basic concepts of standard theory of computation had been developed and their fundamental importance had been stressed by Church's Thesis, a lot of work has been done in generalizing the classical approaches to non–classical object domains. First attempts were based on enumerations and numberings of algebraic structures, we refer to [55, 40, 38, 17]. Y. Moschovakis [51, 52] introduced notions of computability by generalizing the principles of generating primitive-recursive and partial-recursive arithmetic functions. E. Engeler [15] introduced a programming approach to algorithmic properties of structures. Finally, H. Friedman [20, 61, 21] contributed a further, rather natural and general framework by means of his generalized Turing algorithms and effective definitional schemes. For a representative view to these endeavours up to the end of the sixties, we refer to [23], in particular to the critical view by G. Kreisel [37]. J. Shepherdson's papers [60, 61, 62] follow the development up to the present.

Detailed presentations of axiomatic approaches were given in the monographs [18, 19]. We also refer to the theory of program schemes, cf. [32, 39, 24], where abstract programs over classes of structures of related signatures are considered. Several relationships between definability of functions by programs, algebraic properties of the underlying structures and dynamic logic were pointed out, see [71, 34, 35]. Ideas of generalized computability were used to develop a logical basis for dealing with geometrical constructions, [59]. Computational geometry [54] is also based on generalized models of computation. Finally, the several theories of effective analysis and type 2 computability should be mentioned in this context, even if their paradigm of computation by approximation differs considerably from our point of view. Related surveys and discussions can be found in [3, 72, 73, 36].

In some sense, our model is a modification of Friedman's generalized Turing algorithms. Moreover, for structures of finite signatures, it is equivalent to a uniform version of Friedman's effective definitional schemes. In contrast to almost all former approaches however, we explicitly consider (the computability of) string functions instead of functions of some fixed arity over the given universe.

The treatment of string functions is necessary from practical demands in order to get a uniform computation device, and it just gives the opportunity to define time complexity in a BSS–like style. So, like Asser's paper [1] did for classical recursion theory some decades

ago, the present paper tries to stress the importance and usefulness of string processing, now for computability over general structures. We remember that classical complexity theory (like theory of computability, too, as far as it is based on the model of Turing machine) does not immediately deal with functions or decision problems over the natural numbers or the integers. More precisely, it considers the digital encodings of those functions or problems. This means, it deals with strings over finite alphabets which possibly represent numbers. The "genuine" complexity of number problems (considered with respect to strings of numbers) has been scarcely investigated so far, cf. [46, 29].

In order to deal explicitly with strings, the computation device has necessarily to be equipped with facilities for string handling. It must be able to prolongate a current string by a given element of the structure, or to delete elements of the current string. The acceptance of these demands leads to the advantage that some non-trivial suppositions of recursion theoretic results and proof techniques in many of the former approaches, like ω–richness, existence of pairing functions or structurality, become superfluous under rather weak assumptions within our framework. For example, natural numbers can be represented by configurations of our programs, the pairing of strings is easily done, and the evaluation of terms is straightforward.

We notice that crucial ideas of our model of computation with respect to structures of finite signatures, in particular the treatment of P-NP questions, have already been outlined and used by J. Goode [27] and B. Poizat [53]. This confirms the naturalness of the approach we are going to present.

This paper is organized as follows. In Sections 2–4, our model of computation is introduced, several examples are given, the basic types of programs are defined and their relationships to classical notions as well as to the BSS setting are discussed. Moreover, the restriction to so–called bipotent structures is justified. Section 5 establishs connections to a modification of Friedman's effective definitional schemes and characterizes nondeterministic computations by means of projections of deterministic ones. Sections 6 and 7 deal with variants of recognizability for sets of strings and with their interdependences on each other and on properties of the underlying structures. Recognizability is also characterized by means of suitable modifications of Friedman's schemes. Section 8 presents basic results on universal programs over structures of finite signatures, whereas Section 9 characterizes the class of general structures over which universal programs with related properties exist. In Section 10, we show the mutual independence of the existence of universal functions and of m–complete sets, respectively, over a structure.

2 The Model of Computation

$\mathbb{N}_+$ denotes the set of all positive integers. By an *algebraic structure*, we mean a quadruple

$$\mathcal{S} = \langle\ S\ ;\ (C_i : i \in I_C)\,;\ (R_i : i \in I_R)\,;\ (F_i : i \in I_F)\ \rangle\,,$$

where S is a nonempty set, called the *universe* of $\mathcal{S}$; $(C_i : i \in I_C)$ is the (possibly empty) family of *base constants*, i.e. $C_i \in S$, for all $i \in I_C$; the family $(R_i : i \in I_R)$ gives the *base relations*, thus, $R_i \subseteq S^{k_i}$ with some *arity* $k_i \in \mathbb{N}_+$, for all $i \in I_R$; and, analogously, $F_i : S^{l_i} \longrightarrow S$ are the (total) *base functions*, each with some arity $l_i \in \mathbb{N}_+$.

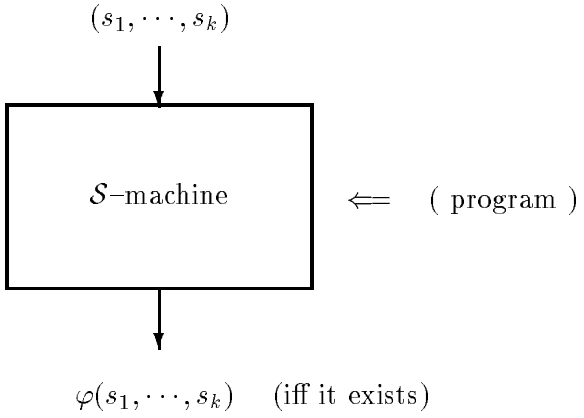
The triple $\sigma = \langle\ I_C\ ;\ (k_i : i \in I_R)\,;\ (l_i : i \in I_F)\ \rangle$ is called the *signature* of $\mathcal{S}$. It is said to be *finite* resp. *countable* if all the *index sets*, I_C, I_R, I_F, are finite resp. countable.

We specify some examples:

$\mathcal{B} = \langle\ \{0,1\}\ ;\ 0,1\ ;\ =\ ;\ +\ ,\ -\ ,\ \cdot\ ,\ /\ \rangle\ ,$	the binary field;
$\mathcal{N} = \langle\ \mathbb{N}\ ;\ 0\ ;\ =\ ;\ succ\ \rangle\ ,$	the Peano structure of natural numbers;
$\mathcal{A} = \langle\ \mathbb{N}\ ;\ 0,1\ ;\ =\ ;\ +\ ,\ \cdot\ \rangle\ ,$	the structure of elementary arithmetic;
$\mathcal{Z} = \langle\ \mathbb{Z}\ ;\ 0,1\ ;\ \leq\ ;\ +\ ,\ -\ ,\ \cdot\ \rangle\ ,$	the ordered ring of integers;
$\mathcal{R} = \langle\ \mathbb{R}\ ;\ 0,1\ ;\ \leq\ ;\ +\ ,\ -\ ,\ \cdot\ ,\ /\ \rangle\ ,$	the ordered field of reals;
$\mathcal{C} = \langle\ \mathbb{C}\ ;\ 0,1\ ;\ =\ ;\ +\ ,\ -\ ,\ \cdot\ ,\ /\ \rangle\ ,$	the field of complex numbers;
$\mathcal{G} = \langle\ G\ ;\ e\ ;\ =\ ;\ \cdot\ ,\ ^{-1}\ \rangle\ ,$	an arbitrary group;
$\mathcal{V} = \langle\ V\ ;\ \mathbf{0}\ ;\ =\ ;\ (\sigma_r : r \in \mathbb{R})\ ,\ +\ \rangle\ ,$	a linear vector space (let $\sigma_r(x) = r \cdot x$);
$\mathcal{R}_{lin} = \langle\ \mathbb{R}\ ;\ 0,1\ ;\ =\ ;\ (\mu_r : r \in \mathbb{R})\ ,\ +\ \rangle\ ,$	the reals as linear space ($\mu_r(x) = r \cdot x$);
$\mathcal{R}_{lin,\leq} = \langle\ \mathbb{R}\ ;\ 0,1\ ;\ \leq\ ;\ (\mu_r : r \in \mathbb{R})\ ,\ +\ \rangle\ ,$	the ordered reals as linear space;
$\mathcal{R}_{sc} = \langle\ \mathbb{R}\ ;\ 0,1\ ;\ =\ ;\ (\mu_r : r \in \mathbb{R})\ \rangle\ ,$	the reals with scalar multiplication only.

The division is here always assumed to be a total operation: let $s/0 = 0$. The first seven examples are structures of finite signatures, the remaining four have infinite signatures.

Fig. 1:



Computability in $\mathcal{S}$, that means of functions $\varphi : S^k \rightarrowtail S$ with some arity k, has been considered by many authors, cf. [15, 31, 20, 61, 59, 34, 35, 21]. It also plays a fundamental role in the theory of program schemes, see [39, 24, 71]. Figure 1 illustrates the underlying basic idea of so–called *finite algorithmic procedures*. The input tuple is given to a "machine" which works according to a certain program and yields a result equal to the value of the

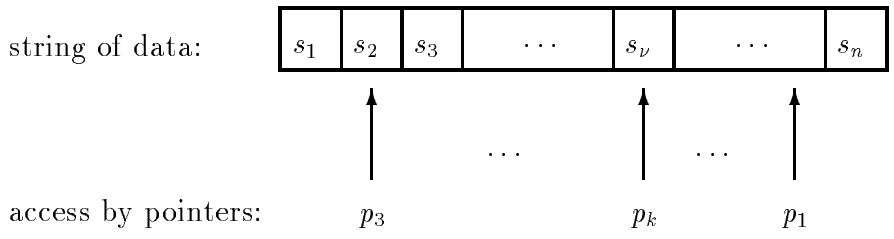
function if and only if the function is defined on that input. Such a (deterministic) Program is a sequence $\mathcal{P} = \langle\, B_0\,,\, B_1\,,\, \cdots\,,\, B_l\,\rangle$ of instructions B_λ which act on finitely many of the *variables* (registers) $x_0\,,\, x_1\,,\, x_2\,,\, \cdots$. Usually, the instructions can be

assignments:	$x_j := C_i$	$(i \in I_C)$;
	$x_j := F_i\,(x_{j_1}, \cdots, x_{j_{l_i}})$	$(i \in I_F)$;
branchings:	*if* $R_i\,(x_{j_1}, \cdots, x_{j_{k_i}})$ *then goto* λ_1	$(i \in I_R)$;
stops:	*halt*.	

Even if the length of input strings is fixed, it can be useful to admit arbitrarily many variables. Practical computation problems, however, very often deal with arbitrarily long sequences of input and output elements. To give some examples, we refer to the solution of systems of equations, say over the reals, to vector and matrix operations over fields, to sorting problems, or to the many problems of computational geometry [54], like the computation of the convex hull of a set of points and others. Thus, we have to consider the computability *over* $\mathcal{S}$, that means the computability of string functions $\varphi : S^* \succ\!\!\!\rightarrow S^*$. The basic idea shown in Figure 1 remains unchanged essentially, but input and output are strings now. To enable the program to have access to arbitrarily many variables, some kind of indirect addressing is needed. This can be implemented by means of special counting variables, as it has been considered by Friedman and subsequently done by several authors as well as in the BSS setting. One could alternatively allow operands of the form x_{x_j} and treat the content of a variable x_j as an address. Both of the variants are based on the availability of natural numbers.

Our approach avoids such a direct reference to natural numbers. We use finitely many pointers which act like the heads of a Turing machine on the current string of data. Moreover, the empty string is not admitted, since we want to avoid the use of a special blank symbol which would have to be added to the elements of the structure. For an illustration of data handling by our machine, see Figure 2.

Fig. 2:



The basic data structure of our $\mathcal{S}$–machine is the set S^+ of all non–empty finite sequences (*strings*) of elements from the universe S. Thus, $S^* = S^+ \cup \{\Lambda\}$, with the *empty string* Λ which is not allowed to occur in the course of our computations. The *concatenation* $w \cdot w'$ or ww' of strings $w = s_1 s_2 \cdots s_n$ and $w' = s'_1 s'_2 \cdots s'_{n'}$ is defined as usual: $w \cdot w' = s_1 s_2 \cdots s_n s'_1 s'_2 \cdots s'_{n'}$. The *length* of a string gives the number of (occurrences of) elements in it, $length(w) = n$.

An $\mathcal{S}$–program will use finitely many *pointer variables* p_j $(1 \le j \le k)$ which point to elements of the current string w, i.e. they have values from $\{1, 2, \cdots, length(w)\}$.

We have three types of *atomic pointer expressions*, namely

$$\begin{array}{l} p_j = p_{j'}\,, \\ r\text{–}end(p_j)\,, \\ l\text{–}end(p_j) \quad (\text{for } 1 \le j, j' \le k)\,. \end{array}$$

In some step, $p_j = p_{j'}$ is true iff both pointer variables have the same current value, whereas $r\text{–}end(p_j)$ resp. $l\text{–}end(p_j)$ are true iff p_j points to the right resp. left end of the current string.

Data variables are of the form " $p_j\uparrow$ " [1], for $1 \le j \le k$; their current values are those elements from S to which the p_j point. *Data terms* are inductively defined to be either data variables or constants C_i $(i \in I_C)$, or to have the form $F_i(t_1, \cdots, t_{l_i})$, with a base function F_i, $i \in I_F$, and data terms $t_1, \cdots, t_{l_i}$.

Atomic data expressions are either equations " $t_1 = t_2$ " with data terms t_1 and t_2, or predicative expressions " $R_i(t_1, \cdots, t_{k_i})$ ", with $i \in I_R$ and data terms $t_1, \cdots, t_{k_i}$.

An $\mathcal{S}$*–program* is a finite sequence

$$\mathcal{P} = (B_0; B_1; \cdots; B_l)\,, \quad l \in I\!N,$$

of unconditional or conditional instructions B_λ, $0 \le \lambda \le l$.
Unconditional instructions are of the following types, where $1 \le j \le k$.

- assignments: " $p_j\uparrow := t$ ", with a data term t ;
- pointer moves: " $r\text{–}move(p_j)$ " or " $l\text{–}move(p_j)$ " ;
- append instructions: " $r\text{–}app(p_j)$ " or " $l\text{–}app(p_j)$ " ;
- delete instructions: " $del(p_j)$ " ;
- stop instructions: " *halt* " ;
- jumps: " $goto(m_0, \cdots, m_n)$ ", with $n, m_0, \cdots, m_n \in I\!N$;
- guess instructions: " $guess(p_j\uparrow)$ ".

A *conditional instruction* has the form

" *if* `Cond` *then* `Inst` "

with an unconditional instruction `Inst` and an atomic (pointer or data) expression `Cond`.

The meanings of assignments, pointer moves and stop instructions are nearly straightforward. Assume that a stop instruction is repeated ad infinitum. Moreover, a pointer move is performed only if the pointer would not leave the current string by that move, otherwise it doesn't cause an action. If the pointer p_j occupies the right resp. left end of the current string, the append instruction causes an enlargement of the string (to the right resp. left)

[1] We shall sometimes use the quotation marks " $\cdots$ " for a better differentiation between constituents of the object language and of the metalanguage, respectively. Remark that the desciptions of the first ones may also contain metavariables.

by one place which has to be filled with the former rightmost resp. leftmost element, and the pointer p_j has to take this position in the following step. If the pointer doesn't occupy the corresponding end of the current string, the append instruction has no effect.

The delete instruction causes an action only if the current string has a length ≥ 2 and if the corresponding p_j points to the right or left end of the string. Then this element has to be removed, and all pointers placed there take the new end element as their positions in the next step. A jump instruction causes a jump to one of the instructions whose indices are given in the list of goal labels, $(m_0, \cdots, m_n)$. In all the other cases of instructions (excepted the stops which call themselves), after having performed them, the program control continues with the next instruction of the program. A guess instruction replaces the value of the corresponding data variable by an arbitrary element of the universe S. Finally, the instruction `Inst`, within some conditional instruction as given above, has to be performed iff the condition `Cond` holds with respect to the current values of the involved pointer and data variables, otherwise the program goes to the next instruction.

More precisely, an operational semantics of $\mathcal{S}$–programs $\mathcal{P}$ can be defined by means of the notion of *$\mathcal{P}$–configuration*.

A *$\mathcal{P}$–configuration* is a $(k+1)$–tuple

$$\kappa = (w, \lambda, \sigma_1, \cdots, \sigma_k)$$

with $w \in S^+, \lambda \in I\!N, 1 \leq \sigma_1, \cdots, \sigma_k \leq \mathit{length}(w)$. It gives the current string w, the content λ of the instruction counter, and the current values of the pointers, $\mathit{val}_\kappa(p_j) = \sigma_j \, (1 \leq j \leq k)$. The values of data variables are $\mathit{val}_\kappa(p_j\!\uparrow) = w[\sigma_j]$ (i.e. the σ_j-th element of the string w). On this basis, the values of pointer expressions and of data terms and data expressions with respect to κ can straightforwardly be defined.

The change of configuration by the execution of one step of a program $\mathcal{P}$ is specified according to the above sketched meanings of instructions. We assume that the last instruction B_l is just the only stop instruction and that all the goal labels (of jumps) occurring in $\mathcal{P}$ belong to the set $\{0, \cdots, l\}$ (otherwise, the call to a non–existing instruction should cause a stop). Thus, to every $\mathcal{P}$–configuration $\kappa = (w, \lambda, \sigma_1, \cdots, \sigma_k)$ obtained from some *initial configuration* $(w_0, 0, 1, \cdots, 1)$ by finitely many steps of the program, an instruction B_λ of the program is assigned, and κ is a *stop configuration* of $\mathcal{P}$ iff $\lambda = l$.

We omit the formal definition of the relation $\vdash_{\mathcal{P}}$, where $\kappa \vdash_{\mathcal{P}} \kappa'$ means that the configuration κ' is obtained from κ by executing one step of program $\mathcal{P}$. For example, a jump "*goto*$(m_0, \cdots, m_n)$" causes $n+1$ successors κ' of the current configuration κ, whereas a guess instruction causes *card*(S) successor configurations. Let $\overset{*}{\vdash}_{\mathcal{P}}$ denote the reflexive and transitive hull of the relation $\vdash_{\mathcal{P}}$, i.e.,

$$\kappa \overset{*}{\vdash}_{\mathcal{P}} \kappa' \quad \text{iff} \quad \begin{array}{l} \text{there are an } m \in I\!N \text{ and } \mathcal{P}\text{–configurations } \kappa_0, \kappa_1, \cdots, \kappa_m \\ \text{such that } \kappa = \kappa_0, \kappa' = \kappa_m, \text{ and } \kappa_i \vdash_{\mathcal{P}} \kappa_{i+1}, \text{ for } 0 \leq i < m. \end{array}$$

Finite or infinite sequences $(\kappa_0, \kappa_1, \kappa_2, \cdots)$ of configurations such that $\kappa_i \vdash_{\mathcal{P}} \kappa_{i+1}$ are called *$\mathcal{P}$–computations*.

Let φ be a string relation from S^+ into S^+, i.e., $\varphi \subseteq S^+ \times S^+$, we write $\varphi : S^+ \rightarrow\!\!\!\!\!\rightarrow S^+$. We shall say that the program $\mathcal{P}$ *computes* the relation φ iff

$$\varphi = \{(w, w') : \text{there is a stop configuration } \kappa' = (w', l, \sigma_1, \cdots, \sigma_k) \text{ such that } (w, 0, 1, \cdots, 1) \overset{*}{\vdash}_{\mathcal{P}} \kappa' \}.$$

For a set $W \subseteq S^+$, we say that $\mathcal{P}$ *recognizes* W iff

$W = \{w$: there is a finite $\mathcal{P}$–computation which starts with $(w, 0, 1, \cdots, 1)$ and terminates with some stop configuration $\}$.

In other words, the recognizable sets over $\mathcal{S}$ are the domains of computable relations or the *halting sets* of $\mathcal{S}$–programs.

A set $W \subseteq S^+$ is called *decidable* iff both W and $S^+ \setminus W$ are recognizable by $\mathcal{S}$–programs.

An $\mathcal{S}$–program is said to be *deterministic* (briefly: D–program) if it does not contain a guess instruction and, moreover, all its jump instructions have just one goal label. Deterministic programs compute only single–valued relations, i.e. (partial) functions. A program is *nondeterministic of the first kind* or *binarily nondeterministic* (N_1–program) if it does not contain a guess instruction. An arbitrary program is also said to be *nondeterministic of the second kind* or *totally nondeterministic* (N_2–program). In the straightforward meaning, we shall speak of deterministic computability, N_i–computability ($i = 1, 2$) and use the related notations with respect to recognitions or decisions.

3 Examples, Quasiprograms, Church's Thesis

In the sequel, programs and parts of programs will also simply be written as sequences of instructions separated each from the other by semicolons. For a better understanding, we sometimes include comments of the form "$\{\cdots \quad \cdots\}$" in programs. Moreover, to facilitate the design and to improve the readability of programs, one can use some almost self–explanatory *meta–notations* for [parts of] programs. They always stand for [parts of] $\mathcal{S}$–programs in the strong sense of definition. So they play the role of *macros* well–known from machine–oriented programming. Their use within recursion theory has been demonstrated in [33, 63].

Sometimes it is convenient to allow the *empty instruction* " ". We shall use *symbolic labels* $L_0, L_1, L_2, \cdots$ instead of indices of instructions as goal labels within jumps. Of course, the corresponding goal instructions must be marked by those labels in the meta–notation. In contrast to our agreement that just the last instruction in $\mathcal{S}$–programs is the stop instruction, in meta–notations we allow the "*halt*" at arbitrary places. Also the use of boolean combinations (generated by the functors "*not*", "*and*" and "*or*") of atomic pointer or data expressions as conditions within conditional instructions is justified. Indeed, every such meta–instruction can straightforwardly be transferred into a sequence of genuine $\mathcal{S}$–instructions with the intended meaning. Moreover, conditional instructions using pointer expressions like "$p_{j_1} < p_{j_2}$", "$p_{j_1} \leq p_{j_2}$", "$p_{j_1} = p_{j_2} + 7$" and also pointer assignments like "$p_{j_1} := p_{j_2} + 5$" can easily be translated into genuine program parts.

Constructs like *if–then–else* can be used, too. For example,

if `Cond` *then* `Inst1` *else* `Inst2` *endif*

can equivalently be replaced by

if `Cond` *then* *goto*(L_1') ;
`Inst2` ; *goto*(L_2') ;
L_1': `Inst1` ;
L_2': ;

with new labels L_1', L_2'. Here `Inst1` and `Inst2` could also be sequences of [meta–] instructions. Analogously, *while–do* or *repeat–until* constructs are admissible.

Thus, for writing programs, the facilities of a high–level language can be applied in the form of meta–notations.

The instructions "*l-app*(p_j)" can be avoided in computations of relations. Instead of this, the current string could be prolongated to the right by one element, and then each element of the former string (and all pointers except the j-th one) can be shifted by one place to the right. Assignments like "$p_{j_1}\uparrow := p_{j_2}\uparrow$" are used to copy elements from one place to another in the current string. It is left to the reader to write a program part performing this idea. Analogously, the application of the instructions "*del*(p_j)" can be avoided always if the j-th pointer occupies the left end of the current string. Instead of that instruction, the elements of the string (and the other pointers) can be shifted to the left by one place. We shall say that a program *works with fixed left end of the strings* if it avoids both the kinds of instructions, the latter for pointers on the left end of the current string only. Obviously, this corresponds to Turing machines with one–sided infinite tapes. We have shown

Lemma 3.1 *Every (D–, N_1– or N_2–) computation of a relation or recognition or decision of a set can be performed by a program working with fixed left end of the strings.* □

By definition, every deterministic program is also nondeterministic of the first or second kind. Furthermore, any unconditional jump instruction

$$goto(L_0, \cdots, L_n)$$

with $n > 1$ can equivalently be replaced by a sequence of *binary* jumps:

$$\begin{array}{l} goto(L_0, L_0') \ ; \\ L_0' : goto(L_1, L_1') \ ; \\ \qquad \vdots \\ L_{n-3}' : goto(L_{n-2}, L_{n-2}') \ ; \\ L_{n-2}' : goto(L_{n-1}, L_n) \ ; \end{array}$$

where $L_1', \cdots, L_{n-1}'$ are symbolic labels which don't occur at other places of the (meta–notation of the) program. Jumps in conditional instructions can analogously be replaced by binary ones.

If the universe S contains only one element, the guess instructions doesn't cause any change of the current string or pointer positions. Thus, it acts deterministically and can equivalently be omitted or replaced by the empty instruction.

If $card(S) \geq 2$, a binary jump " $goto(L_1, L_2)$ " can equivalently be replaced by a program part of the form

$$\begin{array}{l} L_0' : \ r\text{–}move(p_{k+1}) \ ; \\ if \ not(\ r\text{–}end(p_{k+1}) \ ) \ then \ goto(L_0') \ ; \\ r\text{–}app(p_{k+1}) \ ; \ guess(p_{k+1}\uparrow) \ ; \\ if \ p_1\uparrow = p_{k+1}\uparrow \ then \ goto(L_1') \ else \ goto(L_2') \ ; \\ L_1' : del(p_{k+1}) \ ; \ goto(L_1) \ ; \\ L_2' : del(p_{k+1}) \ ; \ goto(L_2) \ ; \end{array}$$

where L_0', L_1', L_2' are new labels which don't occur somewhere else in the program, and p_{k+1} is a new pointer variable. So we have

Lemma 3.2 *On structures with at least two elements, nondeterminism of the first kind can be simulated by means of guess instructions and deterministic instructions only.* □

Over the structure $\mathcal{N}$ of natural numbers, the instruction "$guess(p_j\uparrow)$ " can equivalently be replaced by

$$\begin{array}{l} p_j\uparrow := 0 \ ; \\ L_0' : \ goto(L_1', L_2') \ ; \\ L_1' : \ p_j\uparrow := p_j\uparrow + 1 \ ; \ goto(L_0') \ ; \\ L_2' : \quad ; \end{array}$$

Thus, N_2–programs over $\mathcal{N}$ are not more powerful than N_1–programs. The analogue does not hold, however, over the ordered field $\mathcal{R}$ of real numbers. Indeed, the function $\varphi_{\sqrt{\cdot}}(w) = \sqrt{abs(w[1])}$ is computed by the N_2–program

L_0: *if not*(*r–end*(p_2)) *then goto*(L_0) ; {p_2 to the right end}
r–app(p_2) ;
guess($p_2\uparrow$) ;
if $p_1\uparrow \leq 0$ *then* $p_1\uparrow := (-1) * p_1\uparrow$; {$p_1\uparrow := abs(p_1\uparrow)$}
L_1 : *if* $p_2\uparrow * p_2\uparrow \neq p_1\uparrow$ *then goto*(L_1) ; {infinite cycle}
L_2 : *del*(p_1) ; *if* $p_1 \neq p_2$ *then goto*(L_2) ; {delete the input w}
halt .

Starting with a string consisting of rational numbers, any N_1–program over $\mathcal{R}$ yields strings of rational numbers only. Thus, the function $\varphi_{\sqrt{\cdot}}$ cannot be computed by such a program. Therefore, N_2–programs over $\mathcal{R}$ are strictly more powerful than N_1–programs, with respect to the computation of functions. Using Tarski's [70, 14] method of effective quantifier elimination, one can show, however, that every set of strings over $\mathcal{R}$, which is recognizable by an N_2–program, can even be recognized by a deterministic program, cf. Propositions 7.1 and 7.5 below.

Many authors, like Friedman, Kfoury and BSS, allow the use of arbitrary elements of the universe as constants in their programs. This seems to be quite common with respect to the RAM model over the integers. But in this case, any constant i can be considered as an abbreviation of the term "$1+\cdots+1$" (i times 1) if $i > 0$, and of a term "$(-1)+\cdots+(-1)$" if $i < 0$.

Following the differentiation made by Moschovakis [51, 52], in this paper we shall strictly distinguish between $\mathcal{S}$*–programs*, where only the base constants of the structure are allowed to occur as direct operands, and the so–called $\mathcal{S}$*–quasiprograms* which are analogously defined but allowing arbitrary elements of the universe as direct operands. Those will be denoted as *quasiconstants* in this context. The concepts of computability, recognizability and decidability are straightforwardly applied to quasiprograms.

For example, over structures $\mathcal{S}$ with finite or countable signatures there are only countably many $\mathcal{S}$–programs. On the other hand, if the universe contains uncountably many elements, we have uncountably many $\mathcal{S}$–quasiprograms and even uncountably many decidable sets of strings. Indeed, any singleton $\{w\}$, $w = s_1 \cdots s_n \in S^+$, can be recognized by the following $\mathcal{S}$–quasiprogram.

if $p_1\uparrow \neq s_1$ *or r–end*(p_1) *then goto*(L_0);
r–move(p_1);
if $p_1\uparrow \neq s_2$ *or r–end*(p_1) *then goto*(L_0);
r–move(p_1);
$\vdots$
if $p_1\uparrow \neq s_{n-1}$ *or r–end*(p_1) *then goto*(L_0);
if $p_1\uparrow \neq s_n$ *or not*(*r–end*(p_1)) *then goto*(L_0);
halt ;
L_0: *goto*(L_0) { infinite cycle } .

By a slight modification, one obtains a quasiprogram recognizing the complement $S^+ \setminus \{w\}$.

More amazing is the fact that every subset of $I\!N$ is $\mathcal{R}$–quasidecidable, since its characteristic function with respect to $I\!N$ can be encoded by (the binary representation of) a

real number. Thus, since the elementary theory of every structure of finite signature can straightforwardly be encoded by a set of natural numbers, it is always $\mathcal{R}$–quasidecidable.

The following lemma shows how the computability of relations by quasiprograms can be characterized by the computability by means of programs.

For a relation $\varphi : S^+ \rightrightarrows S^+$ and a (possibly empty) string $w_0 \in S^*$, let the relation $\varphi_{\langle w_0 \rangle}$ be defined by

$$\varphi_{\langle w_0 \rangle} = \{(w, w') : (w_0 \cdot w, w') \in \varphi\} .$$

For a function φ, we simply have $\varphi_{\langle w_0 \rangle}(w) \cong \varphi(w_0 \cdot w)$ [2].

Lemma 3.3 *A relation $\psi : S^+ \rightrightarrows S^+$ is computable by a ($D-$, N_1- or N_2-) quasiprogram over $\mathcal{S}$ iff there are a relation $\varphi : S^+ \rightrightarrows S^+$ and a string $w_0 \in S^*$ such that $\psi = \varphi_{\langle w_0 \rangle}$ and φ is computable by an $\mathcal{S}$–program of the same type.*

To prove the direction "$\longrightarrow$", let $\mathcal{P}$ be an $\mathcal{S}$–quasiprogram which computes ψ. Without loss of generality, we suppose that $\mathcal{P}$ works with a fixed left end of the strings. Assume, it uses the quasiconstants $s_1, \cdots, s_n \in S$, $n \in I\!N$, and the pointer variables $p_1, \cdots, p_k$.

Let $w_0 = s_1 \cdots s_n$, and $\mathcal{P}_0$ be the following program.

$r\text{–}move(p_1);\quad \cdots \quad ; r\text{–}move(p_k);$
$\vdots \qquad\qquad\qquad\qquad \{n \text{ lines}\}$
$r\text{–}move(p_1);\quad \cdots \quad ; r\text{–}move(p_k);$
$r\text{–}move(p_{k+2});\quad \cdots \quad ; r\text{–}move(p_{k+n}) ; r\text{–}move(p_{k+n+1}) ;$
$r\text{–}move(p_{k+3});\quad \cdots \quad ; r\text{–}move(p_{k+n+1}) ;$
$\vdots$
$r\text{–}move(p_{k+n}) ; r\text{–}move(p_{k+n+1}) ;$
$r\text{–}move(p_{k+n+1}) ;$
$\widehat{\mathcal{P}} ;$
$del(p_{k+1}) ;$
$\vdots \qquad\qquad\qquad\qquad \{n \text{ lines}\}$
$del(p_{k+1}) ;$

where $\widehat{\mathcal{P}}$ is obtained from $\mathcal{P}$ by replacing the quasiterms "s_i" by the terms "$p_{k+i}\uparrow$", and by replacing the pointer expressions "$l\text{–}end(p_j)$" by "$p_j = p_{k+n+1}$".

In a preprocessing stage, program $\mathcal{P}_0$ transfers any starting configuration of the form $(w_0 \cdot w, 0, 1, \cdots, 1)$ into a configuration $(w_0 \cdot w, 0, n+1, \cdots, n+1, 1, 2, \cdots, n, n+1)$, and then it holds $(w, \lambda, \sigma_1, \cdots, \sigma_k) \vdash_{\mathcal{P}} (w', \lambda', \sigma'_1, \cdots, \sigma'_k)$ iff $(w_0 \cdot w, c + \lambda, \sigma_1, \cdots, \sigma_k, 1, \cdots, n, n+1) \vdash_{\mathcal{P}_0} (w_0 \cdot w', c + \lambda', \sigma'_1, \cdots, \sigma'_k, 1, \cdots, n, n+1)$, for every $w, w' \in S^+$, where $c = nk + \frac{n(n+1)}{2}$ is the number of instructions of the preprocessing part of $\mathcal{P}_0$. Finally, after the simulation of program $\mathcal{P}$ has finished, in a postprocessing stage, $\mathcal{P}_0$ deletes the prefix w_0 of the current string.

Thus, we have $\psi = \varphi_{\langle w_0 \rangle}$ if ψ and φ are the relations computed by $\mathcal{P}$ and $\mathcal{P}_0$, respectively. If $\mathcal{P}$ is deterministic, N_1 and N_2, respectively, then $\mathcal{P}_0$ is of that type, too.

[2] "$\cong$" denotes the *conditional equality*, i.e., both sides are simutaneously either undefined or defined, and in the latter case the equality holds.

To prove the other direction of Lemma 3, let $w_0 = s_1 \cdots s_n \in S^*$ and $\varphi : S^+ \rightarrowtail\!\!\!\!\!\rightarrow S^+$ be computed by some program $\mathcal{P}_0$. It is left to the reader to give a quasiprogram $\mathcal{P}$ which uses the elements $s_1, \cdots, s_n$ as quasiconstants and works as follows. Starting with some input string $w \in S^+$, $\mathcal{P}$ first generates the string $w_0 \cdot w$, and then it simulates the program $\mathcal{P}_0$ on the w–part of the string. For the relation ψ computed by $\mathcal{P}$, it holds $\psi = \varphi_{\langle w_0 \rangle}$. □

The analogue of the lemma holds also for the recognition or decision of sets of strings.

Finally, one shows easily

Lemma 3.4 *The classes of all D–, N_1–, N_2–computable resp. quasicomputable string functions are closed under composition.* □

The explanations and examples given so far should enable the reader to understand the informal descriptions of programs in the following sections. We close the section with summarizing the relationships between our computation model and Turing's resp. the BSS machine model and proposing a generalization of Church's thesis.

For a finite structure, all elements of which are base constants, $\mathcal{S} = \langle S; S; \cdots; \cdots \rangle$ with $card(S) \in I\!N_+$, the notions of N_1– and N_2–computability coincide. Moreover, a relation $\varphi : S^+ \rightarrowtail\!\!\!\!\!\rightarrow S^+$ is deterministically, N_1– and N_2–computable, respectively, iff it is deterministically resp. nondeterministically computable in the classical sense by some Turing machine working over the alphabet $S \cup \{\flat\}$, where $\flat \notin S$ denotes the blank symbol. In the deterministic case, this means that φ is a partial–recursive word function, cf. [1]. For further relationships to classical computability, see the next section.

Over the ordered field of reals, $\mathcal{R}$, our notion of deterministic computability by quasiprograms coincides with the computability by a deterministic BSS machine. The concept of nondeterminism used by BSS corresponds to our nondeterminism of the second kind. Binary nondeterminism over the reals (also called *digital* or *weak* nondeterminism) has been considered in [12, 13, 27] in connection with polynomially bounded complexity classes.

Generalized Church's Thesis: A string function $\varphi : S^+ \rightarrowtail\!\!\!\!\!\rightarrow S^+$ is $\mathcal{S}$–computable if and only if it is *intuitively computable* over the structure $\mathcal{S}$, i.e., computable by a human being or a machine that

- works sequentially (performing serielly a sequence of elementary steps),
- uses a potentially infinite storage accessible by means of devices like pointers (maybe the fingers of a human being) which are stepwise shiftable and connected with read–write devices,
- and that can generate the base constants and perform the base functions and base relations (inclusively identity " = "), for any given arguments from S.

To formulate the analogue for quasicomputability, one has to suppose additionally the ability of generating arbitrary elements of the universe.

We don't want to discuss this thesis in detail. It should be justified by the whole of this paper. For a detailed generalization of Gandi's principles [22] to computability over arbitrary structures, the reader is referred to [62]. We restrict ourselves to the following two remarks.

By allowing term equations as atomic data expressions, i.e., as tests within conditional instructions, we actually study the computability over the structure *with identity*. This

is justified by the majority of the examples we are thinking of. Moreover, it seems to be appropriate as long as we also allow copy instructions like "$p_{j_1}\uparrow := p_{j_2}\uparrow$" without any scruple. Finally, instead to present the possibly boring maze of several cases and subcases of computability, we prefer to follow first one main stream which includes the essential examples of algebraic structures.

The stepwise shiftable pointers enable the programs to use always the structure $\mathcal{N}$ of natural numbers in an encoded form. Indeed, a number n can be represented by a pointer with the distance n from the left end of the current string, and the counter operations on $\mathcal{N}$ correspond to the shifting of that pointer. By our opinion, the implicite availability of the natural numbers is not a disadvantage of the model, but a cononical consequence of using something like pointers on a linear, potentially infinite storage. Roughly speaking: who cannot count, shouldn't try to compute string functions.

4 Relationships to Classical Computability, Bipotency

An element $s \in S$ is said to be ($\mathcal{S}$–)*constructible* if the total constant function φ_s, with $\varphi_s(w) = s$, for all $w \in S^+$, is deterministically $\mathcal{S}$–computable. If the structure contains at least one base constant, this condition is equivalent to the existence of a ground term over $\mathcal{S}$ with the value s. A string $w_0 \in S^+$ is called *constructible* if it consists of constructible elements only. One could equvalently require that the total constant function φ_w with the value w is D–computable.

A structure $\mathcal{S}$ is called *constructive* if all elements of the universe are constructible. This is the case iff the concept of (deterministic) $\mathcal{S}$–computability coincides with that of (deterministic) $\mathcal{S}$–quasicomputability. Of course, this concept of constructibility is not relevant, since trivial, with respect to quasicomputability.

Proposition 4.1 *Let A be a finite set of $\mathcal{S}$–constructible elements, where* $\mathrm{card}(A) \geq 2$. *Then every partial recursive function $\varphi : A^+ \rightarrowtail A^+$ is deterministically $\mathcal{S}$–computable.*

To sketch the basic ideas of the proof, let $A = \{a_1, \cdots, a_m\}$, and, for $1 \leq \mu \leq m$, let $\mathcal{P}_\mu$ be a deterministic program working with fixed left end of the strings and computing the constant function φ_{a_μ}.

One easily writes a D–program $\mathcal{D}_1$ which, for any input string $w \in S^+$, computes the string $a_1 \cdots a_m \cdot w$. It could work as follows. For $\mu = m, m-1, \cdots, 1$, the given string $a_{\mu+1} \cdots a_m \cdot w$ is transferred into $a_{\mu+1} \cdots a_m \cdot w \cdot s_0$, where s_0 is the first element of string w. Then, by simulating $\mathcal{P}_\mu$ on the input string s_0 to the right of $a_{\mu+1} \cdots a_m \cdot w$, the string $a_{\mu+1} \cdots a_m \cdot w \cdot a_\mu$ is computed. Finally, this is transferred into $a_\mu a_{\mu+1} \cdots a_m \cdot w$.

Starting with the result $a_1 \cdots a_m \cdot w$ of $\mathcal{D}_1$, let the D–program $\mathcal{D}_2$ enter an infinite cycle if $w \notin A^+$. If $w \in A^+$, $\mathcal{D}_2$ may compute the string $a_1 \cdots a_m \cdot code_2(w)$, where the encoding is defined by

$$code_2(a_{i_1} a_{i_2} \cdots a_{i_n}) = a_1 a_2^{i_1} a_1 \; a_1 a_2^{i_2} a_1 \; \cdots \; a_1 a_2^{i_n} a_1,$$

for $n \in \mathbb{N}_+{}^+$, $i_1, \cdots, i_n \in \{1, \cdots, m\}$. Here and in the sequel, let s^i denote the string consisting of i copies of the element s.

Since the word function φ is supposed to be partial recursive, there is a deterministic (1–head) Turing machine which computes φ in the classical sense using some alphabet $B \supseteq A \cup \{\flat\}$, where $\flat$ is the blank symbol. Then one easily obtains a Turing machine $\mathcal{M}$ with the alphabet $\{a_1, a_2, \flat\}$, where $\flat$ is the blank symbol, which computes the function

$$\varphi_{code_2} = \{ (code_2(w), code_2(w')) : \; (w, w') \in \varphi \}$$

in such a way that it never shifts the left end of the current string, that the head never leaves the nonempty part of the tape inscription by more than one place, and that a non–blank letter on the tape can be replaced by the blank symbol only at the right end of the current string.

Such a Turing machine can be simulated by a D–program $\mathcal{D}_3$ which, starting with $a_1 \cdots a_m \cdot code_2(w)$, computes some $a_1 \cdots a_m \cdot code_2(w')$ iff $\varphi(w) = w'$.

Finally, we need a D–program $\mathcal{D}_4$ performing the postprocessing by which some $a_1 \cdots a_m \cdot code_2(w')$ is transferred into the string w'. Now the composition of the programs $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$ yields a D–program $\mathcal{D}$ which computes the function φ. $\square$

We remark that the constructibility of the elements of A is essentially used in the proof, since the programs $\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$ really need the prefix $a_1 \cdots a_m$ of the current strings.

Proposition 4.2 *Let $f : I\!N_+ \succ\!\!\rightarrow I\!N_+$ be a partial recursive (arithmetic) function. Then the function $\varphi_f : \{s^n : s \in S,\, n \in I\!N_+\} \succ\!\!\rightarrow \{s^n : s \in S,\, n \in I\!N_+\}$, defined by $\varphi_f(s^n) \cong s^{f(n)}$, for $s \in S$ and $n \in I\!N_+$, is deterministically $\mathcal{S}$–computable.*
If s is a constructible element of the structure $\mathcal{S}$ and $f : I\!N_+ \succ\!\!\rightarrow I\!N_+$ is a partial recursive (arithmetic) function, then the function $\psi_{f,s} : \{s\}^+ \succ\!\!\rightarrow \{s\}^+$ defined by $\psi_{f,s}(s^n) \cong s^{f(n)}$, for $n \in I\!N_+$, is deterministically $\mathcal{S}$–computable.

To prove the first part of the proposition, we remember Minsky's [50] result that every partial recursive arithmetic function f can be computed by a 3–counter machine $\mathcal{M}$ which, starting with the counter contents $(n, 0, 0)$ (i.e. the first counter contains n, the second and third one are empty), finally halts iff $f(n)$ is defined, and then its counter contents are $(f(n), 0, 0)$.

After testing if the input string belongs to $\{s^n : s \in S,\, n \in I\!N_+\}$ (and entering an infinite cycle otherwise), such a 3–counter machine $\mathcal{M}$ can easily be simulated by a D–program $\mathcal{P}$, where the counter contents (n_1, n_2, n_3) are represented by an $\mathcal{P}$–configuration $(s^n, \lambda, n_1, n_2, n_3)$, with $n = max(n_1, n_2, n_3)$, and λ is the index of an instruction corresponding to the current state of the counter machine.

Now the second part of Proposition 4.2 follows, since, for a constructible element s, the set $\{s\}^+$ can easily be recognized. □

The conversions of Propositions 4.1 and 4.2 don't hold. Over countably infinite universes like $I\!N$, already a unary base functions, (which is trivially computable) could be non–recursive.

A structure $\mathcal{S}$ is said to be *bipotent* if it contains at least two constructible elements r_0, r_1. Examples of bipotent structures are the number domains as they have been specified in Section 2.

Over bipotent structures, our type of program allows us to use *tracks* within the current string $w = s_1 s_2 \cdots s_n$, for example by working with some $\tilde{w} = r_{i_1}\, s_1\, r_{i_2}\, s_2 \cdots r_{i_n}\, s_n$ ($i_1, i_2, \cdots, i_n \in \{0, 1\}$) instead of w. Then on the track of even–numbered places the processing of the current string can be performed, whereas the track of odd–numbered places can be used for auxiliary computations (within the alphabet $\{r_0, r_1\}$) or to store auxiliary marks for the main process. One easily writes a D–program performing the corresponding preprocessing and postprocessing, i.e. computing the functions φ_{pre} and φ_{post}, where

$$\varphi_{pre}(s_1 s_2 \cdots s_n) = r_0 s_1 r_0 s_2 \cdots r_0 s_n \ ,$$

$$\varphi_{post}(r_{i_1}\, s_1\, r_{i_2}\, s_2 \cdots r_{i_n}\, s_n) = s_1 s_2 \cdots s_n \ .$$

We remark that a similar technique can always be applied if the input string contains or only allows to compute two distinct elements r_0 and r_1.

Bipotent structures also allow a rather simple *pairing of strings*. We define

$$pair(\, s_1 \cdots s_n \,,\, s'_1 \cdots s'_{n'} \,) \;=\; r_0 s_1 r_0 s_2 \cdots r_0 s_n\, r_1\, s'_1 r_0 s'_2 \cdots r_0 s'_{n'} \,.$$

Obviously, the set $\{pair(w_1, w_2) : w_1, w_2 \in S^+\}$ is D–decidable. There are D–programs which compute the components w resp. w' of a given string $pair(w, w')$. Conversely, if the

strings w and w' are available in the course of some computation, the string $pair(w, w')$ can deterministically be put here.

Examples of non–bipotent structures are groups $\mathcal{G}$ or vector spaces $\mathcal{V}$, cf. Section 2. Here the neutral elements e resp. $\mathbf{0}$ are sterile. A vector space becomes bipotent if the vectors of some basis are added as base constants.

An element s of a structure $\mathcal{S}$ is said to be *$\mathcal{S}$–sterile* if $\varphi(\{s\}^+) \subseteq \{s\}^+$, for every $\mathcal{S}$–computable function φ. In other words, if s is the only element of which the input consists, no program is able to generate any other element.

One easily shows that, for $\mathcal{S} = \langle S, (C_i : i \in I_C), (R_i : i \in I_R), (F_i : i \in I_F)\rangle$, an element s is $\mathcal{S}$–sterile if and only if $C_i = s$, for all $i \in I_C$, and $F_i(s, \cdots, s) = s$, for all $i \in I_F$.

Proposition 4.2 holds also for sterile elements. Now we are going to sketch some further details connected with sterility. They will give some justification of the later restriction to bipotent structures.

The *domain of sterility* of an arbitrary structure $\mathcal{S}$ is defined by

$$Ster(\mathcal{S}) = \{s^i : i \in I\!N_+ \quad \text{and } s \text{ is } \mathcal{S}\text{–sterile } \} .$$

Lemma 4.1 *Let $\mathcal{S}$ be a structure of finite signature, or let $\mathcal{S}$ have at least one base constant. Then $Ster(\mathcal{S})$ is decidable. Moreover, there is a deterministically $\mathcal{S}$–computable function φ_{diff} such that, for every $w \in S^+ \setminus \mathrm{Ster}(\mathcal{S})$, there are elements r_1, r_2 with $r_1 \neq r_2$ and $\varphi_{diff}(w) = r_1 r_2$.*

If the structure is of finite signature, the first assertion follows from the characterization of sterility given above. For strings of the form $w = s_1^i\, s_2\, w'$ with $i \in I\!N_+$, $s_1 \neq s_2$, and some possibly empty string w', let $\varphi_{diff}(w) = s_1 s_2$. For $w = s^i$ with some non–sterile element s, let $\varphi_{diff}(w) = s\, s'$, where s' is the first element different from s in the concatenation of the sequences $(C_i : i \in I_C)$ and $(F_i(s, \cdots, s) : i \in I_F)$ with respect to some fixed linear ordering in $I_C \cup I_F$. For $w \in Ster(\mathcal{S})$, let $\varphi_{diff}(w)$ be undefined. One easily shows that φ_{diff} is deterministically $\mathcal{S}$–computable.

If a structure (of infinite signature) has two different base constants, it is bipotent, and the assertions hold obviously.

Now let all the base constants coincide with some $C \in S$. We consider two cases. If C is not $\mathcal{S}$–sterile, we have $Ster(\mathcal{S}) = \emptyset$, and a function φ_{diff} is easily computed. If C is $\mathcal{S}$–sterile, then $Ster(\mathcal{S}) = \{C^i : i \in I\!N_+\}$. This set is obviously decidable. For $w \notin Ster(\mathcal{S})$, there is some first element $s \neq C$ in w, and one can take $\varphi_{diff}(w) = C \cdot s$. □

Sterile elements s_1 and s_2 are said to be *equivalent* if, for every base relation R_i of $\mathcal{S}$, it holds

$$R_i(s_1, \cdots, s_1) \quad \text{iff} \quad R_i(s_2, \cdots, s_2) .$$

Thus, equivalent sterile elements cannot be distinguished from each other by $\mathcal{S}$–programs. Let $[s]$ denote the class of sterile elements which are equivalent to s. If $\mathcal{S}$ is of finite signature or has a base constant, there are only finitely many equivalence classes. The following proposition characterizes the deterministic $\mathcal{S}$–computability on the domain of sterility.

Proposition 4.3 *Let $\mathcal{S}$ be a structure of finite signature or with at least one base constant. To every deterministically $\mathcal{S}$–computable string function φ, there is a system $\{f_{[s]} : s \text{ is } \mathcal{S}\text{–sterile } \}$ of partial recursive arithmetic functions $f_{[s]} : I\!N_+ \rightarrowtail I\!N_+$ such that, for every*

sterile element s and every $n \in \mathbb{N}_+$, it holds

$$\varphi(s^n) \cong s^{f_{[s]}(n)} .$$

Conversely, if a function φ with $dom(\varphi) \subseteq Ster(\mathcal{S})$ is defined in that way, for some system $\{f_{[s]} : s$ is $\mathcal{S}$–sterile $\}$ of partial recursive functions, then φ is $\mathcal{S}$–computable.

The first assertion holds generally, since the behaviour of deterministic $\mathcal{S}$–programs on strings of sterile elements can be simulated by deterministic Turing machines, and since $\mathcal{S}$–programs cannot distinguish between equivalent sterile elements.

To prove the second part of the proposition for a structure of finite signature, let a program work as follows, on an input string $s^n \in Ster(\mathcal{S})$. By successively testing all conditions $R_i(s, \cdots, s)$, $i \in I_R$, it finally reaches a branch which corresponds to the equivalence class $[s]$. Then the string $s^{f_{[s]}(n)}$ can be computed analogously to the proof of Proposition 4.2.

If a structure with some base constant C is not bipotent, C is the only sterile element, and the assertion follows by Proposition 4.2. □

To characterize $\mathcal{S}$–computability outside the domain of sterility, let $\bar{\mathcal{S}}$ be obtained from $\mathcal{S}$ by adjuncting two further constants. More precisely, if $\mathcal{S} = \langle S, (C_i : i \in I_C), (R_i : i \in I_R), (F_i : i \in I_F)\rangle$, where $0, 1 \notin I_C$ without loss of generality, then we consider a structure $\bar{\mathcal{S}} = \langle S \cup \{C_0, C_1\}, (C_i : i \in I_C \cup \{0,1\}), (\bar{R}_i : i \in I_R), (\bar{F}_i : i \in I_F)\rangle$, such that $C_0, C_1 \notin S$, and R_i and F_i are restrictions of $\bar{R}_i$ and $\bar{F}_i$, respectively.

Obviously, $\bar{\mathcal{S}}$ is always a bipotent structure. Outside $Ster(\mathcal{S})$, $\mathcal{S}$–computability can be characterized by means of $\bar{\mathcal{S}}$–computability:

Proposition 4.4 *Let $\mathrm{dom}(\varphi) \subseteq S^+ \setminus Ster(\mathcal{S})$, for a partial function φ on a structure $\mathcal{S}$ of finite signature or with at least one base constant. Then φ is deterministically $\mathcal{S}$–computable iff it is deterministically $\bar{\mathcal{S}}$–computable.*

Given an $\mathcal{S}$–program computing φ, one obtains an $\bar{\mathcal{S}}$–program by adding a preprocessing part which becomes cyclic if the input string contains the new constants C_0 or C_1.

Conversely, let φ be $\bar{\mathcal{S}}$–computable by some program $\mathcal{P}$. To compute φ, an $\mathcal{S}$–program can work as follows. On inputs $w \in Ster(\mathcal{S})$, it enters an infinite cycle. Inputs $w = s_1 \cdots s_n \in S^+ \setminus Ster(\mathcal{S})$ are transformed into $r_0 r_1 \cdot w$, with two distinct elements r_0, r_1 of $\mathcal{S}$. This is possible by Lemma 4.1. Then, in the latter case, the program constructs the string $r_0 r_1\, r_0 s_1 r_0 s_2 \cdots r_0 s_n$. Now, by means of the prefix $r_0 r_1$ and the track of odd–numbered places, the working of the $\bar{\mathcal{S}}$–program $\mathcal{P}$ can be simulated in such a way that the elements s of $\mathcal{S}$ are encoded by substrings $r_0 s$, whereas the new constants C_0 and C_1 are encoded by $r_1 r_0$ and $r_1 r_1$, respectively. Here the first elements r_0 or r_1 belong to the track of odd–numbered places. □

Proposition 4.5 *Over a structure $\mathcal{S}$ of finite signature or with at least one base constant, a partial function $\varphi : S^+ \rightarrowtail S^+$ is deterministically $\mathcal{S}$–computable iff it can be represented in the form*

$$\varphi = \varphi_1 \cup \varphi_2 ,$$

where $dom(\varphi_1) \subseteq Ster(\mathcal{S})$, $dom(\varphi_2) \subseteq S^+ \setminus Ster(\mathcal{S})$, and φ_1 and φ_2 have the properties given in Propositions 4.3 and 4.4, respectively.

This follows from Lemma 4.1 and Propositions 4.3 and 4.4. □

Over many structures, the number of pointers used by programs can universally be bounded. Let $deg(\mathcal{S})$ denote the maximal arity of base functions or relations of structure $\mathcal{S}$ if these arities are bounded, and $deg(\mathcal{S}) = \omega$ otherwise.

Proposition 4.6 *If the structure $\mathcal{S}$ has no sterile element, then, to every deterministic $\mathcal{S}$–program $\mathcal{P}$, there is a deterministic $\mathcal{S}$–program $\mathcal{P}'$ computing the same string function and using at most $max(deg(\mathcal{S}), 2)$ pointer variables.*
On a structure $\mathcal{S}$ of finite signature or a structure with at least one base constant, every deterministically computable function can be computed by a deterministic $\mathcal{S}$–program using at most $max(deg(\mathcal{S}), 3)$ pointer variables.

Of course, this holds analogously for nondeterministic programs of both kinds.

On a bipotent structure or outside the domain of sterility of an arbitrary structure, every computation of a program $\mathcal{P}_2$ can be simulated by a program $\tilde{\mathcal{P}}_2$ using only $max(deg(\mathcal{S}), 2)$ pointers if it works as follows. It stores the current string w in a first track and uses sufficiently further tracks to mark the current positions of the pointer variables of $\mathcal{P}_2$ and some auxiliary labels if needed during some step of simulation. This can be done by means of two constructible elements r_0, r_1 or by means of the elements r_1, r_2 from $\varphi_{diff}(w) = r_1 r_2$, for the input string w, according to Lemma 4.1.

For example, to simulate an assignment $p_j\uparrow := F_i(p_{j_1}\uparrow, \cdots, p_{j_{l_i}}\uparrow)$, the current values of the data variables $p_{j_1}\uparrow, \cdots, p_{j_{l_i}}\uparrow$ are copied into an auxiliary string of length l_i, say to the right of the current string. Two pointer variables are suffucient to do this. Then the first l_i pointers of the simulating program $\tilde{\mathcal{P}}_2$, $q_1, \cdots, q_{l_i}$, are placed on the elements of that string, and the assignment $q_1\uparrow := F_i(q_1\uparrow, \cdots, q_{l_i}\uparrow)$, is performed. Finally, the result is copied from the first position of the auxiliary string to the current position of p_j within the simulated $\mathcal{P}_2$–computation, and the auxiliery string is erased then.

More complicated assignments and other instructions of program $\mathcal{P}_2$ can analogously be simulated. Thus, the first assertion of the proposition has been proved.

In the second case, on the domain of sterility of a structure $\mathcal{S}$, a program $\mathcal{P}_1$ computes a string function φ obtained from some system $\{f_{[s]} : s \text{ is } \mathcal{S}\text{–sterile}\}$ of partial recursive functions $f_{[s]}$ according to Proposition 4.3. By the technique used in the proof of Proposition 4.2, one can show that every such function φ on $Ster(\mathcal{S})$ can be computed by a 3–pointer program $\tilde{\mathcal{P}}_1$. The decision of $Ster(\mathcal{S})$ according to Lemma 4.1 can be performed by means of two pointer variables. This completes the proof of the second part of Proposition 4.6. □

Agreement: In the remaining part of this paper, all structures $\mathcal{S}$ we are dealing with are assumed to be bipotent. Let r_0 and r_1 always denote two fixed $\mathcal{S}$–constructible elements.

This restriction is justified by Propositions 4.3 - 4.5 which analogously hold for nondeterministic computations as well as for recognitions and decisions. The structures of infinite signature we are preferably interested in satisfy the agreement, too.

5 Computation Paths, Umeds, Nondeterminism

To every deterministic $\mathcal{S}$–program $\mathcal{P} = \langle B_0; B_1; \cdots; B_l \rangle$, $l \in \mathbb{N}$, the possible sequences of indices of instructions performed during $\mathcal{P}$–computations can be arranged in a binary *computation tree* $\mathcal{T}_\mathcal{P}$. This is a finite or infinite directed rooted tree whose vertices are nonempty strings $v \in \{0, 1, \cdots, l\}^+$.

Without loss of generality, we suppose that every conditional instruction of $\mathcal{P}$ is a conditional jump. Then $\mathcal{T}_\mathcal{P}$ is inductively defined as follows.

> The string $v_0 = 0$ is the root of $\mathcal{T}_\mathcal{P}$.
> A vertex $v\lambda$, $v \in \{0, 1, \cdots, l\}^*$, $\lambda \in \{0, 1, \cdots, l\}$ has no son iff $B_\lambda =$ "*halt*" ;
> it has a son $v\lambda\lambda'$ iff
>
> - B_λ is an unconditional assignment, pointer move, append or delete instruction, and $\lambda' = \lambda + 1$;
> - B_λ is an unconditional jump "*goto*(m)", and $\lambda' = m$; or
> - B_λ is a conditional jump "*if* `Cond` *then goto*(m)", and $\lambda' = \lambda + 1$ or $\lambda' = m$.

By our general supposition that B_l is the only stop instruction and all jumps have destinations within the program, all leaves of $\mathcal{T}_\mathcal{P}$ have the final letter l. A vertex $v\lambda$ has two sons iff the instruction B_λ is a conditional jump with a jump destination $m \neq \lambda + 1$. The rooted paths of $\mathcal{T}_\mathcal{P}$ correspond to $\mathcal{P}$–computations starting with initial configurations. Every input string $w \in S^+$ determines exactly one maximal rooted path in $\mathcal{T}_\mathcal{P}$. We call it the *computation path* of w. It is finite iff w belongs to the halting set of program $\mathcal{P}$.

On the other hand, to every vertex $v = \lambda_0\lambda_1 \cdots \lambda_k$ of $\mathcal{T}_\mathcal{P}$ ($k \in \mathbb{N}, \lambda_0, \cdots, \lambda_k \in \{0, 1, \cdots, l\}$), we can assign the set W_v of all input strings whose $\mathcal{P}$–computations start with the instruction sequence $B_{\lambda_0}, \cdots, B_{\lambda_k}$. If these sets W_v are restricted to input strings of some fixed length, we obtain sets which are representable by quantifier–free first–order expressions.

Let $S^n = \{w \in S^+ : \mathit{length}(w) = n\}$. A set $W \subseteq S^+$ is said to be *booleanly first–order representable* (briefly: bfo–representable) if it can be written in the form

$$W = \{s_1 \cdots s_n : \mathcal{S} \models H(s_1, \cdots, s_n)\},$$

for some quantifier–free first–order expression $H = H(x_1, \cdots, x_n)$ over the (language of) structure $\mathcal{S}$. The latter means that the expression H contains at most the variables $x_1, \cdots, x_n$ and is a boolean combination of term equations and atomic relational expressions over $\mathcal{S}$. In particular, it has to be parameter–free. $\mathcal{S} \models H(s_1, \cdots, s_n)$ denotes the validity of the expression H in the structure $\mathcal{S}$ if the variables $x_1, \cdots, x_n$ are replaced by the individuals $s_1, \cdots, s_n \in S$.

We shall say that the expression $H = H(x_1, \cdots, x_n)$ *represents* the set W if the above equation holds.

Lemma 5.1 *For each $n \in \mathbb{N}_+$ and each vertex v of $\mathcal{T}_\mathcal{P}$, the set $W_v \cap S^n$ is bfo–representable. Moreover, if the structure $\mathcal{S}$ is of finite signature, there is an (in the classical sense) effective procedure which, for any given pair (n, v), generates an expression representing $W_v \cap S^n$.*

The basic idea of the proof is simple. Shepherdson [61] describes it as *following the action of $\mathcal{P}$ symbolically.* If a vertex $v\lambda$ has two sons, $v\lambda\lambda_1'$ and $v\lambda\lambda_2'$ in $\mathcal{T}_{\mathcal{P}}$, one of them corresponds to the validity of the test condition of instruction B_λ, the other one to the validity of its negation. If we have only data expressions as conditions within the program, by following the rooted path to vertex v within the computation tree, the data variables $p_j\uparrow$ in the terms or test conditions of the program can successively be replaced by terms depending on individual variables $x_1, \cdots, x_n$ which represent the elements of the input string. More precisely, these terms take the current values of the data variables whenever the $x_1, \cdots, x_n$ are replaced by the corresponding elements of the input string $w = s_1 \cdots s_n$.

The only problem is caused by the fact that we also admitted pointer expressions as test conditions within the program $\mathcal{P}$. If the length n of the input is fixed or only known, however, one can keep track the pointer moves in the course of $\mathcal{P}$–computations corresponding to some vertex $v\lambda$. If the instruction B_λ depends on a pointer condition now, exactly one of the sons, say $v\lambda\lambda_1'$, corresponds to the validity of that condition. The other son, $v\lambda\lambda_2'$, corresponds to no computation. Thus, we have $W_{v\lambda} \cap S^n = W_{v\lambda\lambda_1'} \cap S^n$, and $W_{v\lambda\lambda_2'} \cap S^n = \emptyset$. Hence the pointer condition of the instruction B_λ can be omitted in describing $W_{v\lambda\lambda_1'} \cap S^n$, and $W_{v\lambda\lambda_1'} \cap S^n$ can be represented by "$\neg(x_1 = x_1)$".

The stepwise construction described above can effectively be performed if we deal with a structure of finite signature. In this case, we can assume that the index sets I_C, I_R, I_F are subsets of $\mathbb{N}$, and the first–order expressions over $\mathcal{S}$ can canonically be encoded by strings over a suitable finite alphabet. For infinite signatures, the above construction is effective, too, if an encoding of the (finitely many) base constants, relations and functions involved in the program is provided. Unfortunately, there is no canonical universal encoding in the general case. □

As we have seen, at any step of a deterministic computation on an input word of length n, all elements of the current string can be represented by $\mathcal{S}$–terms in the variables $x_1, \cdots, x_n$ which correspond to the input elements. Hence this presentation is also possible for the elements of the output strings of the program. Moreover, the lengths of these strings and the term representations of their elements are uniquely determined by the leaf of the computation tree which corresponds to the halting computation yielding that output. Combining this observation with the representation of the sets $W_v \cap S^n$ given obove, we obtain a characterization of computable string functions by means of an adaptation of Friedman's [20] effective definitional schemes.

By a *uniform modified effective definitional schema* (briefly: umeds) over the structure $\mathcal{S}$, we understand an (in the classical sense) effectively computable total function

$$\Sigma : \mathbb{N}_+ \times \mathbb{N}_+ \longrightarrow A^+,$$

where A is a suitable finite alphabet and every $\Sigma(n,k)$ is a conditional expression of the form

$$\text{“ } \textit{if } H(x_1, \cdots, x_n) \textit{ then } (\, t_1(x_1, \cdots, x_n)\,, \cdots, t_m(x_1, \cdots, x_n)\,)\text{”}$$

with some $m \in \mathbb{N}_+$, a bfo–expression H and terms $t_1, \cdots, t_m$, all depending on the individual variables $x_1, \cdots, x_n$.

We say that the umeds Σ *defines* the relation $\varrho_\Sigma : S^+ \rightarrowtail S^+$ given by

$$\varrho_\Sigma = \{(s_1 \cdots s_n \,,\, s'_1 \cdots s'_m \,) : \; n \in I\!N_+ \,, \text{ there is some } k \in I\!N_+ \text{ such that } \Sigma(n,k) = \\ \text{“ } \textit{if } H(x_1, \cdots, x_n) \textit{ then } (t_1(x_1, \cdots, x_n) \,, \cdots, t_m(x_1, \cdots, x_n))\text{”}, \\ \mathcal{S} \models H(s_1, \cdots, s_n) \text{ and } s'_\mu = t_\mu(s_1, \cdots, s_n), \text{ for } 1 \leq \mu \leq m \,\}.$$

Theorem 5.1 *Let $\mathcal{S}$ be a structure of finite signature. A string function $\varphi : S^+ \rightarrowtail S^+$ is deterministically $\mathcal{S}$–computable iff it can be defined by a umeds over $\mathcal{S}$.*

The direction "$\longrightarrow$" holds by the introductory remarks, especially by the idea of following the program's actions symbolically. To prove the opposite direction, let a umeds Σ be given. Since we consider bipotent structures, from Proposition 4.1 it follows that the computation of Σ can be simulated by a deterministic $\mathcal{S}$–program. Indeed, the two $\mathcal{S}$–constructible elements r_0 and r_1 can be used to encode the letters b_i of an arbitrary finite alphabet, for example by the strings $r_0 {r_1}^i r_0$. Hence parameter–free terms and bfo–expressions can be encoded by strings from $\{r_0, r_1\}^+$ in such a way that their values, for given elements to be assigned to the variables, are computable by an $\mathcal{S}$–program.

To compute the function φ defined by Σ, for some input string $w = s_1 \cdots s_n$, a deterministic $\mathcal{S}$–program computes the sequence $\Sigma(n,1), \Sigma(n,2), \cdots$ up to obtaining a conditional expression “ *if* $H(x_1, \cdots, x_n)$ *then* $(\, t_1(x_1, \cdots, x_n) \,, \cdots \,, t_m(x_1, \cdots, x_n) \,)$” , where $\mathcal{S} \models H(s_1, \cdots, s_n)$; then take $\varphi(w) = t_1(s_1, \cdots, s_n) \cdots t_m(s_1, \cdots, s_n)$ (this is a string of length m). If the sequence $\Sigma(n,1), \Sigma(n,2), \cdots$ does not yield such a conditional expression, $\varphi(w)$ remains undefined. □

This theorem shows that, with respect to structures of finite signature, our notion of computability of string functions is equivalent to a straigthforward adaptation of Friedman's concept of eds computability for functions of fixed arity. Roughtly speaking, $\mathcal{S}$–computability means classical computability extended by the ability to evaluate terms and boolean first–order expressions over $\mathcal{S}$, for given elements from the universe. Since Friedman's eds computability is widely acknowledged as an acceptable and most general concept of computability in algebraic structures, the theorem gives a further justification of our generalized Church's thesis, cf. Section 3.

By a slight modification of the proof idea sketched above, we obtain a characterization of N_1–computability.

Theorem 5.2 *Let $\mathcal{S}$ be a structure of finite signature. A string relation $\varrho : S^+ \rightarrowtail\!\!\!\!\!\rightarrow S^+$ is $\mathcal{S}$–computable by an N_1–program iff it can be defined by a umeds over $\mathcal{S}$.*

Let be given an N_1–program $\mathcal{P}$ over $\mathcal{S}$ wich computes some string relation ϱ. We suppose that all nondeterministic jumps are binary ones and that all conditional instructions are deterministic. For a string $a = r_{i_1} \cdots r_{i_k} \in \{r_0, r_1\}^+$, let $\mathcal{P}_a$ denote the deterministic $\mathcal{S}$–program obtained from $\mathcal{P}$ by taking the m_1–branch of a nondeterministic jump "*goto*(m_0, m_1)" if it is reached in a k'-th step of working, where $k' \leq k$ and $i_{k'} = 1$, and taking the m_0–branch otherwise (in particular if $k' > k$).

If φ_a denotes the partial function computed by $\mathcal{P}_a$, we obviously have

$$\varrho = \bigcup_{a \in \{r_0, r_1\}^+} \varphi_a \,.$$

Given an input length n, for $\mu = 1, 2, 3, \cdots$ and for all strings $a \in \{r_0, r_1\}^\mu$, one successively generates and uses according to the proof of Theorem 5.1 the computation trees $\mathcal{T}_{\mathcal{P}_a}$ up to the depth μ and puts out the conditional expressions corresponding to the leaves obtained in this way. This procedure computes a umeds defining ϱ.

For the opposite direction, let be given a umeds Σ defining some string relation ϱ. An N_1–program $\mathcal{P}$ computing ϱ can work as follows, on an input string $w = s_1 \cdots s_n$.
For $k = 1, 2, 3, \cdots$, $\Sigma(n, k)$ is computed. Assume it has the form
" *if* $H(x_1, \cdots, x_n)$ *then* $(\, t_1(x_1, \cdots, x_n)\,, \cdots, t_m(x_1, \cdots, x_n)\,)$". If $\mathcal{S} \models H(s_1, \cdots, s_n)$, the program performs a nondeterministic binary choice whether it either stops with the output string $t_1(s_1, \cdots, s_n) \cdots t_m(s_1, \cdots, s_n)$ or continues the computation (with $k := k + 1$). □

Corollary 5.1 *Every N_1–computable function is D–computable.*

Over structures of finite signature, this follows immediately from Theorems 5.1 and 5.2. In the general case, it is obtained by a successive simulation of all the initial segments of the computation paths of an N_1–program by a deterministic one. This can be done analogously to the first part of the previous proof. Instead of generating conditional expressions, only the corresponding actions of the program have to be simulated. □

Theorem 5.2 and the corollary show that the N_1–computability of relations is a natural concept closely related to classical computability. In contrast to this, nondeterminism of the second kind can be much more powerful and seems intuitively to go beyond the scope of natural effectiveness. In Section 3, we already demonstrated the N_2–computability of the square root over $\mathcal{R}$. A general trivial example is given by the maximal string relation $S^+ \times S^+$. It is always N_2–computable if the structure contains at least two elements. On the other hand, for every N_1–computable relation ϱ, the complete image of a string $w \in S^+$, i.e., the set $\{w' : (w, w') \in \varrho\}$, is always finite or countably infinite.

Another characterization of nondeterminism can be obtained by means of projections of deterministic computations. This is widely used in the BSS framework. Here, the nondeterminism of the second kind looks more natural.

Theorem 5.3 *A string relation ϱ over a structure $\mathcal{S}$ is N_2–computable iff there is a D–computable string function φ such that*

$$\varrho = \{(w, w') : \text{ there is a string } a \in S^+ \text{ such that } \varphi(pair(w, a)) = w'\}\,.$$

A string relation ϱ over a structure $\mathcal{S}$ is N_1–computable iff there is a D–computable string function φ such that

$$\varrho = \{(w, w') : \text{ there is a string } a \in \{r_0, r_1\}^+ \text{ such that } \varphi(pair(w, a)) = w'\}\,.$$

To prove this, let $\mathcal{P}$ be an N_2–program computing a string relation ϱ. By Lemma 3.2, we suppose that $\mathcal{P}$ contains no nondeterministic jump. We consider a D–program $\mathcal{P}_0$ which works as follows. First it checks whether the input is an encoding of a pair of strings. If not, $\mathcal{P}_0$ enters a cycle of work, i.e., it yields no result. If yes, let the input have the form $pair(w, a)$, with an 'advise string' $a = s'_1 \cdots s'_{n'}$. Now $\mathcal{P}_0$ simulates the program $\mathcal{P}$ in such a way that when $\mathcal{P}$ has to perform a guess instruction at the k-th time, $\mathcal{P}_0$ takes the element

s'_k as the result of this guessing if $k \leq n'$. If $k > n'$ in this situation, let $\mathcal{P}_0$ take $s'_{n'}$ as the guess result. One easily sees that ϱ is representable as given in the theorem, where φ is the string function computed by $\mathcal{P}_0$.

Conversely, if a D–computable function φ is given, let an N_2–program $\mathcal{P}'$, starting on some input string w, first guess a string a and compute $\varphi(\mathit{pair}(w, a))$ then. The relation ϱ computed by $\mathcal{P}'$ satisfies the first equation in Theorem 5.3 again.

The proof of the second part of the theorem which concerns the N_1–computability is analogous. □

By means of Lemma 3.3, all results of this section can immediately be transferred to quasicomputability. For example, over structures of finite signatures, a string function is D–quasicomputable resp. a string relation is N_1–quasicomputable iff they can be defined by an quasi–umeds (defined like an umeds, but with finitely many elements of the structure which may occur as quasiconstants).

6 Recognizability and Related Concepts

There are many notions of recognizable, semirecursive and recursively enumerable sets, respectively, which all coincide with respect to classical computability. H. Friedman [20] considered some of them as well as related properties of structures and computability. Now we are going to deal with this complex of problems within our framework. In particular, it will turn out in the next two sections that several of the related results by R. Saint John [57, 58] for ordered subrings of the reals can be generalized to arbitrary structures.

Throughout this section, we only treat deterministic programs and computations. Remember also the agreement to consider bipotent structures $\mathcal{S} = \langle S; (C_i : i \in I_C); (R_i : i \in I_R); (F_i : i \in I_F)\rangle$ only, where two $\mathcal{S}$–constructible elements r_0, r_1 are fixed.

As defined in Section 2, by *recognizable* sets of strings, $W \subseteq S^+$, we understand the domains of D–computable string functions $\varphi : S^+ \succ\!\!\!\rightarrow S^+$, $W = dom(\varphi)$. Also the synonymous denotation *halting set* is used. An *output set* W is a range of a D–computable string function φ, $W = ran(\varphi)$.

A set $W \subseteq S^+$ is said to be $(\mathcal{S}$–$)$*enumerable* if it is empty or there are an $\mathcal{S}$–constructible string w_0 and a D–computable (partial) string function ψ such that

$$W = \{\psi^i(w_0) : i \in I\!N\}.$$

Here, ψ^i denotes the i-th iteration of ψ. This representation of W means that it can be exhausted by an $\mathcal{S}$–effective counting process starting with the constructible string w_0. ψ is also called a *successor function enumerating* W.

Lemma 6.1 *Every halting set can be represented as the domain of a D–computable injective string function.*
Every non-empty output set is the range of a D–computable total string function.
Every enumerable set with at least two elements can be represented in the form $W = \{\psi^i(w_0) : i \in I\!N\}$, with a constructible string w_0 and a D–computable injection ψ such that $W = dom(\psi)$.

To prove the first assertion, we define

$$\varphi'(w) \cong pair(\varphi(w), w), \quad \text{for all } w \in S^+ .$$

Then we have $dom(\varphi') = dom(\varphi)$, φ' is injective and D–computable if φ is.

Now let $W = ran(\varphi)$, for a string function φ computed by some D–program $\mathcal{P}$, and $w_0 \in W$. We define a total function $\psi : S^+ \longrightarrow S^+$ by

$$\psi(w) = \begin{cases} \varphi(w') & \text{if } w = pair(w', r_0^k) \text{ for some } w' \in S^+, k \in I\!N_+ , \\ & \quad \text{and the } \mathcal{P}\text{–computation on input } w' \text{ stops after} \leq k \text{ steps}, \\ w_0 & \text{otherwise } (w \text{ isn't a pair of the above form}, \\ & \qquad \text{or } w \text{ has that form, but the } \mathcal{P}\text{–computation is longer}). \end{cases}$$

It holds $ran(\psi) = ran(\varphi) = W$, and ψ is a D–computable total function.

To show the third assertion, let $W = \{\psi^i(w_0) : i \in I\!N\}$, $W \neq \{w_0\}$. To compute a function ψ' with $W = dom(\psi') = \{\psi'^i(w_0) : w \in I\!N\}$, let a D–program $\mathcal{P}'$, on some

input string w, successively perform the computation of $\psi^i(w_0)$, for $i = 0, 1, 2, \cdots$, up to reaching an i_0 such that $w = \psi^{i_0}(w_0)$. Then let it put out the string $\psi'(w) = \psi^{i_0+1}(w_0)$. If $w \neq \psi^i(w_0)$, for all $i \in I\!N$, let $\psi'(w)$ be undefined. For $\{w_0\} \subset W$, φ' is an injection. $\square$

The requirement that the counting function ψ in the representation $W = \{\psi^i(w_0) : i \in I\!N\}$ of an enumerable set must be total, would lead to a properly restricted concept already in classical computation theory (since there exist simple sets).

To give an example of a bipotent structure which owns an output set that is not a range of a D–computable injection, consider

$$\mathcal{E} = \langle\, I\!N\, ;\, 0, 1\, ;\, =\, ;\, E\,\rangle\,,$$

where E is a unary function given by $E(0) = E(1) = E(2) = 0$, and $E(k) = 2$, for all $k \in I\!N \setminus \{0, 1, 2\}$. The set $W = \{2\} \subseteq I\!N^1$ is an output set, as one easily shows. Let φ be an $\mathcal{E}$–computable function with $ran(\varphi) = \{2\}$. If $\varphi(w) = 2$, the input string w must contain an element $k \geq 3$. Let k be the maximal element of w. Then the input string w' obtained from w by replacing k everywhere by the element $k + 1$ yields the same result: $\varphi(w') = 2$. This can be shown analyzing the possible premises of conditional expressions which can occur in an umeds defining φ; the details are left to the reader. Thus, φ is not injective.

Proposition 6.1 *The classes of all recognizable sets, of all output sets and of all enumerable sets, respectively, are closed under pairwise union and intersection.*

To deal with output sets first, let $W_i = ran(\varphi_i)$, for $i = 1, 2$, where the functions φ_i are computed by D–programs $\mathcal{P}_i$. If we put

$$\psi(w) \cong \begin{cases} \varphi_1(w_1) & \text{if } w = pair(w_1, w_2) \\ & \quad \text{and } \varphi_1(w_1), \varphi_2(w_2) \text{ are defined both and equal,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

then it follows $ran(\psi) = W_1 \cap W_2$.

For

$$\psi(w) \cong \begin{cases} w_1 & \text{if } w = r_0 \cdot pair(w', r_0^k) \\ & \quad \text{and } \mathcal{P}_1 \text{ stops on input } w' \text{ after} \leq k \text{ steps with output } w_1, \\ w_2 & \text{if } w = r_1 \cdot pair(w', r_0^k) \\ & \quad \text{and } \mathcal{P}_2 \text{ stops on input } w' \text{ after} \leq k \text{ steps with output } w_2, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

it holds $ran(\psi) = W_1 \cup W_2$.

To compute a function ψ with $dom(\psi) = dom(\varphi_1) \cap dom(\varphi_2)$, let a program first perform the φ_1–computation, where the input string w has to be preserved. This can be done on a special track as explained in Section 4, or simply on a special area separated from the working space of the φ_1–program by an additional pointer. After that, if $\varphi_1(w)$ exists, let the program perform the φ_2–computation on the same input w.

Now let $W_i = dom(\varphi_i)$, for $i = 1, 2$, with functions φ_i computed by D–programs $\mathcal{P}_i$. To compose a D–program computing a function ψ with $dom(\psi) = W_1 \cup W_2$, we use a technique of *alternating stepwise simulation* of both of the programs $\mathcal{P}_1$ and $\mathcal{P}_2$.

To simplify the explanation, we assume that $\mathcal{P}_1$ works with fixed left end of strings, whereas $\mathcal{P}_2$ works analogously with the fixed right–hand end of strings. The program $\mathcal{P}$, in the alternating simulation of the both, uses two disjoint sets of pointers, each corresponding to the pointer set of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. Let the frontier between the both workspaces of simulation be marked by a further pointer p^*. Moreover, the contents of the program counters of $\mathcal{P}_1$ resp. $\mathcal{P}_2$ (i.e., the indices of the instructions to be performed as the next ones) should be encoded, say by the distances of two special pointers, p_1^* and p_2^*, from p^*. To this purpose, the current string has to be sufficiently enlarged, by using tracks or additional pointers. The simulating program stops when one of the programs $\mathcal{P}_1$ and $\mathcal{P}_2$ reaches a stop configuration.

We hope the underlying idea has become clear by the rough description above. Let us stress that the alternating simulation of two fixed programs does not yet require a kind of a universal program (which does not necessarily exist for arbitrary structures of possibly infinite signature, cf. Section 9).

Now let $W_j = \{\psi_j^i(w_j) : i \in I\!N\}$, for $j = 1, 2$, with constructible strings w_1, w_2 and functions ψ_1, ψ_2 computed by $\mathcal{S}$–programs $\mathcal{P}_1$ resp. $\mathcal{P}_2$.

If $W_1 \cap W_2$ is finite, it is enumerable, since it is empty or consists of constructible elements only. If the intersection is infinite, it can successively be generated by alternating stepwise simulation of generating processes of W_1 and W_2 using computations of w_1 and w_2 and the programs $\mathcal{P}_1$ resp. $\mathcal{P}_2$. Herein, the simulating program has to keep track the list of elements of both W_1 and W_2 which are generated so far. Obtaining a new element of W_1 resp. W_2, it has to be checked if this has already been generated as an element of the other set. So we obtain a process of generating $W_1 \cap W_2$ which can be used to compute a first element and a successor function enumerating $W_1 \cap W_2$.

It is easy to generate the union $W_1 \cup W_2$ if one of the sets W_1 and W_2 is finite. If both the sets are infinite, there is a program which generates the union by alternating generation of one further element of W_1 and W_2, respectively, starting with w_1 and w_2. By keeping track the list of all elements generated so far, the program can be modified in such a way that a repeated generation of elements (namely those from $W_1 \cap W_2$) is avoided. □

Now we are going to deal with relationships between the classes of halting sets, output sets and enumerable sets, respectively. For structures of finite signatures, a first attempt has been done under the author's guidance in [2].

Proposition 6.2 *Every enumerable set is recognizable; every recognizable set is an output set.*

The first assertion holds trivially for the empty set and each singleton $\{w_0\}$ consisting of a constructible element w_0. For sets with at least two elements, it has already been noticed in Lemma 6.1.

To prove the second assertion, let $W = dom(\varphi)$, for some D–computable function φ. Then the function ψ defined by

$$\psi(w) \cong \begin{cases} w & \text{if } \varphi(w) \text{ is defined} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

is D–computable too, and we have $W = ran(\psi)$. □

The conversions don't hold generally, as will be shown soon. In particular, the enumerability is a rather special property. One easily sees

Lemma 6.2 *An enumerable set is always finite or countably infinite, and it consists of constructible elements only.* □

If every output set or only every halting set is enumerable over some structure $\mathcal{S}$, this especially holds for the universe S. Now we are going to show that, under the supposition of finite signature, this property holds exactly for the constructive structures.

Proposition 6.3 *The universe S of a structure is enumerable iff the set of all strings, S^+, is enumerable.*

The direction "$\longleftarrow$" can easily be proved. If S is finite and contains only constructible elements (since S^+ does), it is trivially enumerated. If S is infinite, it is enumerated by using the enumeration process for S^+ and skipping all strings of lengths > 1.

The direction "$\longrightarrow$" for finite universes S can be shown by some standard method from classical computation theory. For example, if $card(S) = m$, perform a counting process for the (bijective) m–adic number representation, where the digits correspond to the elements of S (in the order given by the enumeration of S).

If S is infinite, for $k = 0, 1, 2, \cdots$, successively generate all strings of lengths $\leq k$ which at most contain the first k elements of S, with respect to the ordering corresponding to the enumeration. □

Corollary 6.1 *If the universe S is enumerable, then the structure $\mathcal{S}$ is constructive.*

This immediately follows from Lemma 6.2 and Proposition 6.3. □

Proposition 6.4 *A structure of finite signature is constructive iff its universe is enumerable.*

We show the nontrivial direction "$\longrightarrow$". If the structure is constructive, every element s of the universe can be obtained as $s = \varphi(r_0)$, for some D–computable function φ. By Theorem 5.2 (or directly, having a look to our kind of computation), it follows that s can be obtained as the value of some $\mathcal{S}$–term after replacing all variables by r_0. If the structure is of finite signature, there is a standard encoding of $\mathcal{S}$–terms, all these encodings can effectively be generated and the values of the terms (for r_0 assigned to all occurring variables) can be computed. On this basis, one obtains a D–computable successor function enumerating S. □

This proposition stresses that the concept of constructivity introduced in Section 4 descibes a natural and essential class of structures, at least for finite signatures.

The supposition of finite signature is essential for the equivalence of constructivity and enumerability. Indeed, the structure $\mathcal{R}_{lin}$ defined in Section 2 is obviously constructive, but its universe, $\mathbb{R}$, is uncountable, hence it is not enumerable.

Theorem 6.1 *For an arbitrary structure $\mathcal{S}$, the following conditions are equivalent:*

(a) the universe of $\mathcal{S}$ is enumerable;

(b) every halting set over $\mathcal{S}$ is enumerable;

(c) every output set over $\mathcal{S}$ is enumerable.

By Proposition 6.2, we have $(c) \longrightarrow (b)$; $(b) \longrightarrow (a)$ holds trivially.

To show $(a) \longrightarrow (c)$, let $S^+ = \{\varphi^i(w_0) : i \in I\!N\}$, cf. Proposition 6.3, and $W = ran(\varphi)$, where φ is computed by some program $\mathcal{P}$. We consider only the nontrivial case that W is infinite.

There is a D–program $\mathcal{P}^*$ which, for $k = 0, 1, 2, \cdots$, simulates always k steps of the computation of $\mathcal{P}$ on the inputs $w_0, \varphi(w_0), \cdots, \varphi^k(w_0)$, in this succession. This generates an effectively computable sequence consisting of all elements of W ordered by the time at which they are generated as outputs of $\mathcal{P}$, in the course of the simulating procedure $\mathcal{P}^*$. By means of $\mathcal{P}^*$, one easily obtaines a D–computation of a function ψ enumarating W, i.e., $W = \{\psi^i(u_0) : i \in I\!N\}$. In particular, u_0 is that string from W, which is first obtained by program $\mathcal{P}^*$. □

Theorem 6.1 together with Proposition 6.4 gives a good characterization of the structures on which enumerable sets and halting or output sets coincide. Now we are going to deal with the relationships between halting sets and output sets which seem to be more complicated, at least over non–enumerable structures. For example, it is known that over the ordered field of real numbers, $\mathcal{R}$, all output sets are halting sets [48, 11]. Over general structures, this coincidence is equivalent to the closure property of the class of all halting sets under projection.

For a set $W \subseteq S^+$, its *projection* (to the first component) is defined by

$$\Pi(W) = \{w_1 : \text{ there is a } w_2 \in S^+ \text{ such that } pair(w_1, w_2) \in W\}.$$

Proposition 6.5 *The classes of all enumerable sets and of all output sets, respectively, are closed under projection.*

From a representation $W = \{\psi^i(w_0) : i \in I\!N\}$, with a constructible string w_0 and a D–computable function ψ, one obtains a representation $\Pi(W) = \{\widetilde{\psi}^i(\widetilde{w_0}) : i \in I\!N\}$, for $\Pi(W) \neq \emptyset$, by generating all first components of the strings $\psi^i(w_0)$ and avoiding repetitions (cf. the proof of Proposition 6.1).

If $W = ran(\varphi)$, for some D–computable function φ, then it holds $\Pi(W) = ran(\psi \circ \varphi)$, where

$$\psi(w) \cong \begin{cases} w_1 & \text{if } w = pair(w_1, w_2), \text{ for some } w_1, w_2 \in S^+, \\ \text{undefined} & \text{otherwise.} \end{cases}$$ □

Theorem 6.2 *Over arbitrary structures, the following conditions are equivalent:*

(a) the classes of all halting sets and of all output sets, respectively, coincide;

(b) the class of all halting sets is closed under projection;

(c) the class of all halting sets is closed under images of D–computable functions.

From Proposition 6.5, it follows that (a) implies (b).

To show the converse, let $W = ran(\varphi)$, for some D–computable function φ. Then $W = \Pi(\widetilde{W})$, for $\widetilde{W} = \{pair(w_1, w_2) : \varphi(w_2) \cong w_1\}$. Obviously, $\widetilde{W}$ is a halting set. Thus, by means of (b), W is a halting set, too.

The output sets are just the images of halting sets under D–computable functions. Therefore, (a) is equivalent to (c). □

One easily shows that the class of all output sets is always closed under images of computable functions. Analogously, both the halting sets and the output sets are closed under pre-images of computable functions.

For a simple example of a structure owning an output set that is not a halting set, let

$$\mathcal{M} = \langle\, I\!N\, ;\, 0, 1\, ;\, =\, ;\, \cdot \,\rangle,$$

where "$\cdot$" denotes the multiplication.

0 and 1 are the only constructive elements of the universe of $\mathcal{M}$. Hence $I\!N$ is a halting set which is not $\mathcal{M}$–enumerable. The set of square numbers, $\{k^2 : k \in I\!N\}$, is an output set but not a halting set. Indeed, for inputs of length 1, $w = x \in I\!N$, by the possible premises of conditional expressions with the only variable x, the elements from $I\!N \setminus \{0, 1\}$ cannot be separated each from the other.

The halting sets seem to give the most natural basis for dealing with logical recognition or decision problems. Thus, our notion of decidability of sets of strings is defined by means of halting sets, cf. Section 2.

By Propositions 6.1 and 6.3, enumerable sets with enumerable complements can exist only over structures with enumerable universes. For these, the classes of enumerable, halting, and output sets, respectively, coincide however, by Theorem 6.1.

To define decidability in such a way that both the set and its complement have to be output sets, would yield a properly more general concept than our definition by means of halting sets. Indeed, both the set $\{2\}$ and its complement $I\!N^+ \setminus \{2\}$ are output sets over the structure $\mathcal{E}$ defined after the proof of Lemma 6.1. But they are not decidable in our sense.

The following proposition stresses the naturalness of our notion of decidability. We show that it coincides with the notion we would obtain by requiring that the characteristic function of the set is D–computable. For $W \subseteq S^+$, the *characteristic function* is defined by

$$\chi_W^{r_0,r_1}(w) = \begin{cases} r_1 & \text{if } w \in W \\ r_0 & \text{if } w \notin W. \end{cases}$$

Proposition 6.6 *A set of strings, W, is decidable iff its characteristic function, $\chi_W^{r_0,r_1}$, is D–computable.*

If the characteristic function is computable, both W and $S^+ \setminus W$ can easily be represented as domains of computable functions. Conversely, given functions φ_1 and φ_2 with the domains W and $S^+ \setminus W$, respectively, by alternating stepwise simulation like in the proof of Proposition 6.1, one shows the D–computability of the characteristic function. □

Finally, we characterize the projections of decidable sets.

Proposition 6.7 *Over an arbitrary structure, the projections of decidable sets of strings are just the output sets.*

Let $W = ran(\varphi)$, for a string function φ computed by some D–program $\mathcal{P}$. For the set $\widetilde{W} = \{pair(pair(r_0^k, w_1), w_2) : \ \mathcal{P}$ stops on input w_2 after $\leq k$ steps with the output $w_1\}$, we have $W = \mathrm{II}(\widetilde{W})$, and $\widetilde{W}$ is decidable.

Conversely, if $W = \mathrm{II}(\widetilde{W})$, for some decidable set $\widetilde{W}$, then $\widetilde{W}$ is also an output set and W too, by Proposition 6.5. □

By Theorem 6.2 and Proposition 6.7, it seems to be justified to introduce a special notation for structures whose output sets coincide with the halting sets.

A structure $\mathcal{S}$ is said to be *normal* if every output set over $\mathcal{S}$ is $\mathcal{S}$–recognizable.

By Theorem 6.1, every structure with an enumerable universe is normal. The ordered field of real numbers, $\mathcal{R}$, represents an example of a normal non–enumerable structure. By Proposition 6.7 and Theorem 6.2, it holds

Corollary 6.2 *A structure $\mathcal{S}$ is normal iff the $\mathcal{S}$–recognizable sets are just the projections of decidable sets or, equivalently, iff the class of all recognizable sets is closed under projections resp. images of D–computable functions.* □

7 D and N_1 versus N_2, Umeeds and Um[e]ers

By Corollary 5.1, every N_1–computable function is D–computable. One can straightforwardly generalize the notions of recognizability considered in the previous section by allowing N_i–computable functions in the corresponding definitions instead of D–computable only. Thus, we have N_i–*enumerable*, N_i–*recognizable* and N_i–*output sets* $(i = 1, 2)$ now, and the former concepts are also specified by the prefix "D–" if we want to stress the contrast to their nondeterministic analogues.

Corollary 5.1 implies that N_1–enumerable sets are always D–enumerable, that N_1–recognizability coincides with D–recognizability, and that every N_1–output set is a D–output set.

For nondeterminism of the second kind, the situation changes. Of course, every N_2–enumerable set is finite or countable infinite. Let us consider the computability over the structure

$$\mathcal{N}_- = \langle\, I\!N\, ;\, 0, 1\, ;\, =\, ;\, \pi\, \rangle,$$

where π denotes the unary predecessor function (say with $\pi(0) = 0$). For every D–computable function $\varphi : I\!N^1 \rightarrowtail I\!N^1$ (i.e. both the domain and the range consist of strings of length 1 only), it holds $\varphi(n) \leq n$, for all $n \in dom(\varphi) \setminus \{0\}$. Thus, every D–enumerable set $W \subseteq I\!N$ is finite. On the other hand, the successor function $\sigma(n) = n + 1$ is N_2–computable over $\mathcal{N}_-$. Therefore, $I\!N$ is N_2–enumerable.

Comparing the notions of recognizability, we obtain a further characterization of normal structures.

Proposition 7.1 *A structure $\mathcal{S}$ is normal iff every N_2–recognizable set over $\mathcal{S}$ is D–recognizable.*

Let $\mathcal{S}$ be a normal structure. By Theorem 5.3, a set $W \subseteq S^+$ is N_2–recognizable iff it is the projection of a D–recognizable set $\widetilde{W}$. Since the class of (D–) recognizable sets is closed under projection by Theorem 6.2, $W = \Pi(\widetilde{W})$ is D–recognizable, too.

For the converse, one easily shows that every (D–) output set is N_2–recognizable. Thus, the structure is normal if the N_2–recognizable sets are D–recognizable. □

Notice that the field of real numbers is a normal structure (i.e. N_2–recognizability coincides with D–recognizability) over which there are N_2–computable functions (like the square root) which are not D–computable. Therefore, the coincidence of the recognizabilities does not imply equality of the concepts of computability for string functions (by deterministic and N_2–programs, respectively).

In the case of halting and output sets, one could also consider N_i–computable relations instead of functions. By Theorem 5.3, the ranges of N_i–computable relations coincides with the ranges of suitable D–computable functions. The following proposition characterizes the relationships with respect to halting sets of relations.

Proposition 7.2 *Every domain of an N_1–computable relation can be represented as the domain of a D–computable function. Every domain of an N_2–computable relation is the domain of a D–computable function iff the underlying structure is normal.*

The first assertion follows from the presentation of N_1–computable relations according to Theorem 5.3. Given such a presentation of a relation, a deterministic computation with the same domain can be performed by successively trying all advices $a \in \{r_0, r_1\}^+$ in their lexicographic ordering and halting when $\varphi(pair(w,a))$ is defined at the first time, for the input w and some advice a.

The second assertion follows from Proposition 7.1 and the above remarks on output sets and by the observation that the N_2–recognizable sets are just the N_2-output sets, also for relations. □

In classical computation theory, a function is computable iff its graph is recognizable, and the analog holds for relations with respect to nondeterministic programs. Over arbitrary structures, this doesn't remain generally valid.

For a string relation $\varrho \subseteq S^+ \times S^+$, its *graph* is defined to be the following set of strings,

$$graph(\varrho) = \{pair(w,w') : (w,w') \in \varrho\}.$$

Lemma 7.1 *Let $X \in \{D, N_1, N_2\}$. If a string relation ϱ is X–computable, then the set* graph(ϱ) *is X–recognizable.*

Indeed, the set $\{pair(w,w') : (w,w') \in S^+\}$ is (deterministically) decidable, and in order to recognize if some string $pair(w,w')$ belongs to $graph(\varrho)$, an X–program can simply perform the computation of ϱ on the input w and check if the output (if some is obtained) equals w'. □

The converse does not hold for $X \in \{D, N_1\}$, not even on normal structures and for functions. Indeed, the graph of the square root, i.e. the set $\{pair(r^2, r) : r \in I\!R, r \geq 0\}$ is (deterministically) decidable over $\mathcal{R}$, but the function is not N_1–computable. For the nondeterminism of the second kind, one easily shows

Lemma 7.2 *A relation is N_2–computable iff its graph is N_2–recognizable.* □

The validity of the graph property for relations and N_1–computability turns out to be equivalent to the enumerability of the structure.

Proposition 7.3 *The following conditions are equivalent:*

(a) the universe of a structure $\mathcal{S}$ is enumerable;

(b) every relation with an (D– or N_1–) recognizable graph is N_1–computable;

(c) for string relations, the N_2–computability coincides with the N_1–computability;

(d) the relation $S^+ \times S^+$ is N_1–computable.

If the universe is enumerable and the graph of a relation ϱ is N_1–recognizable, the relation can be N_1–computed by successively testing all strings if they are images of some given input with respect to ϱ. Hence we have $(a) \longrightarrow (b)$; similar it follows $(a) \longrightarrow (c)$.

$(b) \longrightarrow (d)$ and $(c) \longrightarrow (d)$ are obvious.

For $(d) \longrightarrow (a)$, notice that the graph of the relation $\varrho = S^+ \times S^+$ is trivially recognizable. If ϱ is N_1–computable, then S^+ can successively be generated by simulating all paths of the computation of ϱ for the input string $w = r_0$. The deterministic simulation of

an N_1–program is done like sketched in the proof of Theorem 5.2. The obtained procedure generating S^+ can straightforwardly be used to establish an enumeration of S^+. $\Box$

Now we generalize the notion of umeds in order to characterize the N_2–computability of relations over structures of finite signature. This is done by admitting a prefix of existential quantifiers in the conditional expressions. We remark that similar concepts have been used by I. Soskov and A. Soskova [64, 65, 69, 68], with respect to computability in enumerated structures. Soskov [64, 66, 67] established close relationships between computability via enumerations and the approaches by Friedman and Moschovakis, respectively. Connections of the latter kind have already been announced 25 years ago by C. Gordon [28].

More precisely, by an *umeeds* (this abbreviates "*uniform modified existentialized effective definitional schema*") over a structure $\mathcal{S}$, we understand an (in the classical sense) effectively computable total function

$$\Sigma : I\!N_+ \times I\!N_+ \longrightarrow A^+,$$

where A is a suitable finite alphabet and every $\Sigma(n,k)$ is an *existentialized conditional expression*, i.e., it is of the form

$$\text{“ } \exists y_1 \cdots \exists y_l \, (\ \textit{if } H(x_1,\cdots,x_n,y_1,\cdots,y_l) \\ \textit{then } \ (\ t_1(x_1,\cdots,x_n,y_1,\cdots,y_l)\,,\ \cdots\ ,\ t_m(x_1,\cdots,x_n,y_1,\cdots,y_l)\ )\ )\text{”}$$

with some $l \in I\!N$, $m \in I\!N_+$, a bfo–expression H and terms $t_1,\cdots,t_m$, all depending on the variables $x_1,\cdots,x_n,y_1,\cdots,y_l$ only.

The meaning of an umeeds within our framework is rather straightforward. One has to notice, however, that both the expression H and the terms $t_1,\cdots,t_m$ belong to the scope of the same quantification of the variables $y_1,\cdots,y_l$ within the existentialized conditional expression.

Thus, we say that the umeeds Σ *defines* the relation $\varrho_\Sigma : S^+ \rightrightarrows S^+$ given by

$$\varrho_\Sigma = \{(\ s_1\cdots s_n\ ,\ s'_1\cdots s'_m\ ) :\ n \in I\!N_+\ \text{, there is some } k \in I\!N_+ \text{ such that } \Sigma(n,k) = \\ \text{“ } \exists y_1 \cdots \exists y_l \, (\ \textit{if } H(x_1,\cdots,x_n,y_1,\cdots,y_l) \textit{ then} \\ (\ t_1(x_1,\cdots,x_n,y_1,\cdots,y_l)\,,\ \cdots\ ,\ t_m(x_1,\cdots,x_n,y_1,\cdots,y_l)\ )\ )\text{”}, \\ \text{and there are elements } \tilde{s_1},\cdots,\tilde{s_l} \in S \text{ such that} \\ \mathcal{S} \models H(s_1,\cdots,s_n,\tilde{s_1},\cdots,\tilde{s_l}) \\ \text{and } s'_\mu = t_\mu(s_1,\cdots,s_n,\tilde{s_1},\cdots,\tilde{s_l}), \text{ for } 1 \le \mu \le m\ \}.$$

Theorem 7.1 *Let $\mathcal{S}$ be a structure of finite signature. A string relation $\varrho : S^+ \rightrightarrows S^+$ is N_2–computable iff it can be defined by a umeeds over $\mathcal{S}$.*

The proof is similar to that of Theorems 5.1 and 5.2. To compute the string relation ϱ_Σ, an N_2–program $\mathcal{P}$ can act on the input string $w = s_1\cdots s_n$ as follows. It chooses nondeterministically a number k and computes the existentialized conditional expression $\Sigma(n,k)$. Assume it has the form given in the definition above. Then the program $\mathcal{P}$ guesses elements $\tilde{s_1},\cdots,\tilde{s_l} \in S$. If $\mathcal{S} \models H(s_1,\cdots,s_n,\tilde{s_1},\cdots,\tilde{s_l})$, let $\mathcal{P}$ output the corresponding string “ $t_1(s_1,\cdots,s_n,\tilde{s_1},\cdots,\tilde{s_l})\cdots t_m(s_1,\cdots,s_n,\tilde{s_1},\cdots,\tilde{s_l})$ ”; otherwise let it enter a cycle of working.

The proof of the converse uses Theorem 5.3. Let be given an N_2–computable string relation ϱ and a deterministic program $\mathcal{P}$ computing a string functions φ such that

$$\varrho = \{(w,w') :\ \text{there is a string } a \in S^+ \text{ with } \varphi(pair(w,a)) = w'\ \}.$$

Without loss of generality, suppose that $\mathcal{P}$ stops only after at least $\bar{n}$ steps, on input strings of length $\bar{n}$. (Otherwise, the program wouldn't be able to visit all elements of the input by its pointers before it stops.)

We describe the way of working of an effective procedure $\mathcal{P}_0$ successively generating all pairs $(\,(n,k)\,,\,\Sigma(n,k)\,)$, for an umeeds Σ such that $\varrho = \varrho_\Sigma$. Given a number $n \in I\!N$, for $\bar{l} = 1, 2, 3, \cdots$ and all $l \in \{1, 2, \cdots, \bar{l}\}$, the procedure $\mathcal{P}_0$ follows symbolically, as described in the proof of Theorem 5.1, the first $\bar{l}$ steps of the action of program $\mathcal{P}$ on inputs represented by $pair(x_1 \cdots x_n, y_1 \cdots y_l)$.

Whenever a terminating computation path of program $\mathcal{P}$ is obtained by the procedure $\mathcal{P}_0$, it puts out the corresponding existentialized conditional expression

$$\text{“ } \exists y_1 \cdots \exists y_l \,(\ \textit{if}\ H(x_1, \cdots, x_n, y_1, \cdots, y_l)$$
$$\textit{then}\ (\ t_1(x_1, \cdots, x_n, y_1, \cdots, y_l)\,,\ \cdots\ , t_m(x_1, \cdots, x_n, y_1, \cdots, y_l)\,)\,)\text{”}$$

within some pair $(\,(n,k)\,,\,\Sigma(n,k)\,)$, for a suitable number k (depending on the history of $\mathcal{P}_0$ up to that moment). Here the expression H characterizes the related computation path of program $\mathcal{P}$, whereas the terms $t_1, \cdots, t_m$ represent the values of the elements of the output string corresponding to that path.

This generating procedure $\mathcal{P}_0$ can easily be modified to a procedure computing the umeeds Σ. It is obvious that $\varrho = \varrho_\Sigma$. □

From Theorems 7.1 and 5.2 and Propositions 6.4 and 7.3, it follows immediately

Proposition 7.4 *A structure $\mathcal{S}$ of finite signature is constructive iff, to every umeeds over $\mathcal{S}$, there is a umeds defining the same string relation.* □

It turns out that the elimination of the existence quantifier in all umeeds over a structure $\mathcal{S}$ is not related to quantifier elimination in the logical sense.

We say that a structure $\mathcal{S}$ *admits effective quantifier elimination* if there is an (in the classical sense) effective procedure which, to every first–order expression $H(x_1, \cdots, x_n)$ over $\mathcal{S}$, yields a quantifier–free expression $\widetilde{H}(x_1, \cdots, x_n)$ equivalent to $H(x_1, \cdots, x_n)$.

The structure of elementary arithmetic, $\mathcal{A}$, is constructive but does not admit effective quantifier elimination, since its first–order theory is undecidable [26]. On the other hand, the ordered field of real numbers, $\mathcal{R}$, admits effective quantifier elimination [70, 14], but the square root represents an N_2–computable function that is not N_1–computable.

If we restrict ourselves to recognizability of sets of strings, however, there is a connection between quantifier elimination and the equivalence of both kinds of nondeterminism, as we are going to show now.

In the following definitions, the brackets "[" and "]" are used as metasymbols to mark optional phrases. To obtain both variants of the sentences, include resp. omit the contents of these brackets everywhere in the corresponding context.

By a *uniform modified* [*existentialized*] *effective recognitional schema* (briefly: *um*[*e*]*ers*) over a structure $\mathcal{S}$ of finite signature, we understand an (in the classical sense) effectively computable total function

$$\Sigma : I\!N_+ \times I\!N_+ \longrightarrow A^+,$$

where A is a suitable finite alphabet, and every $\Sigma(n,k)$ is an [existential] expression of the form

$$\text{“ } [\exists y_1 \cdots \exists y_l]\, H(x_1, \cdots, x_n\, [\,, y_1, \cdots, y_l\,]\,)\text{”},$$

where $[l \in I\!N,]\, m \in I\!N_+$, and H is a bfo–expression depending on the variables $x_1, \cdots, x_n$ $[\,, y_1, \cdots, y_l\,]$ only.

We say that the um[e]eds Σ *defines* the following set of strings over $\mathcal{S}$.

$$W_\Sigma = \{s_1 \cdots s_n : n \in I\!N_+,\ s_1, \cdots, s_n \in S \text{ and there is some } k \in I\!N_+ \text{ such that}$$
$$\Sigma(n,k) = \text{“}\,[\,\exists y_1 \cdots \exists y_l\,]\, H(x_1, \cdots, x_n\,[\,, y_1, \cdots, y_l\,]\,)\text{”},$$
$$\text{and [there are elements } \tilde{s_1}, \cdots, \tilde{s_l} \in S \text{ such that]}$$
$$\mathcal{S} \models H(s_1, \cdots, s_n\,[\,, \tilde{s_1}, \cdots, \tilde{s_l}\,]\,)\,\}.$$

Theorem 7.2 *Over a structure of finite signature, a set of strings is N_1–recognizable iff it can be defined by a umers, and a set is N_2–recognizable iff it is definable by a umeers.*

The N_i–recognizable sets are the domains of N_i–computable functions ($i = 1, 2$). From Theorems 5.2 and 7.1 and the above definitions, it follows that such sets can be defined by umers and umeers, respectively. Indeed, starting from a um[e]eds defining a function φ, one obtains a um[e]ers defining the set $dom(\varphi)$ by erasing both the "*if*" and the whole *then*–part of the [existentialized] conditional expressions.

Conversely, given a um[e]ers defining some set of strings, W, it can be modified to a um[e]eds defining a string function with the domain W. To this purpose, one can always take the term "x_1" representing the first element of the input to define the values of the function in the *then*–parts. □

By this theorem, Corollary 5.1 and Proposition 7.1, it follows

Corollary 7.1 *A structure $\mathcal{S}$ of finite signature is normal iff to every umeers over $\mathcal{S}$ there is a umers defining the same set of strings.* □

Now let the underlying structure $\mathcal{S}$ admit effective quantifier elimination. Then, to every umeers over $\mathcal{S}$, there is effectively constructible a umers defining the same set of strings. By this and Proposition 7.1, it follows

Proposition 7.5 *Every structure of finite signature which admits effective quantifier elimination is normal.* □

As the example of elementary arithmetics shows, the normality of a structure is not equivalent to admitting effective quantifier elimination. We will show, however, that normality is equivalent to a certain weak kind of effective elimination of prefixes of existential quantifiers.

In the following, we use some standard encoding of the deterministic programs over the monoid over a suitable alphabet (well–known from classical theory of computability). For a first–order expression $H = H(x_1, \cdots, x_n)$ over some structure $\mathcal{S}$, the *satisfiability set* of H is defined to be the following set of strings,

$$W_H = \{s_1 \cdots s_n \ :\ \mathcal{S} \models H(s_1, \cdots, s_n)\}.$$

Obviously, the set defined by some um[e]ers equals the union of the satisfiability sets of the expressions generated by that um[e]ers. Recall that an *existential expression* is a first–order expression of the form

$$\exists y_1 \ \cdots \ \exists y_l \ H(x_1, \cdots, x_n, y_1, \cdots, y_l)\,,$$

where $H(x_1, \cdots, x_n, y_1, \cdots, y_l)$ is a bfo–expression over the underlying structure.

Proposition 7.6 *Let $\mathcal{S}$ be a structure of finite signature with at least one base constant. Then $\mathcal{S}$ is normal iff there is an effective procedure which, for every existential expression H, yields a (code of a) program (in the classical sense) that computes a umers defining the satisfiability set of H.*

The proof is based on Corollary 7.1.

Assume that there is given an effective procedure $\mathcal{P}$ yielding, to every existential expression H, a code of a program $\mathcal{P}_H$ that (in the classical sense) computes a umers Σ_H defining the set W_H. Let Σ be an arbitrary umeers over the structure $\mathcal{S}$. By means of the projections π_1 and π_2 of Cantor's pairing function for $\mathbb{N}_+$, we define a umers Σ' by

$$\Sigma'(n,k) = \Sigma_{\Sigma(n,\pi_1(k))}(n,\pi_2(k)) \text{ , for } n,k \in \mathbb{N}_+ .$$

The computability of Σ' follows from classical recursion theory. Σ' defines the same set of strings as Σ, namely the set

$$\bigcup_{n,k\in\mathbb{N}_+} W_{\Sigma'(n,k)} = \bigcup_{n,k_1\in\mathbb{N}_+} \bigcup_{k_2\in\mathbb{N}_+} W_{\Sigma_{\Sigma(n,k_1)}(n,k_2)}.$$

Thus, the structure $\mathcal{S}$ is shown to be normal.

Conversely, let the structure $\mathcal{S}$ be normal. Therefore, to every umeeds there is a umeds defining the same set of strings. Since $\mathcal{S}$ is of finite signature, the existential expressions over $\mathcal{S}$ can biuniquely and effectively in both directions be encoded by strings from $\{r_0,r_1\}^+$. For $w \in \{r_0,r_1\}^+$, let H_w denote the existential expression encoded by w.

We consider the following set of strings,

$$\begin{aligned}\widehat{W} = \{pair(w_1,w_2) :\ & w_1 = s_1\cdots s_n, \text{ for some } n \in \mathbb{N},\\ & \text{and } H_{w_2} \text{ is an existential expression with } n \text{ free variables,}\\ & \text{say } H_{w_2} = H_{w_2}(x_1,\cdots,x_n), \text{ such that } \mathcal{S} \models H_{w_2}(s_1,\cdots,s_n)\ \}.\end{aligned}$$

$\widehat{W}$ is N_2–recognizable over $\mathcal{S}$. Indeed, if some string $pair(w_1,w_2)$ with $w_1 = s_1\cdots s_n$ and $H_{w_2} = H_{w_2}(x_1,\cdots,x_n) = \exists y_1\cdots\exists y_l \widetilde{H_{w_2}}(x_1,\cdots,x_n,y_1,\cdots,y_l)$ is given, then the validity of $H_{w_2}(s_1,\cdots,s_n)$ can be recognized by guessing elements $\widetilde{s_1},\cdots,\widetilde{s_l} \in S$ and proving that $\mathcal{S} \models \widetilde{H_{w_2}}(s_1,\cdots,s_n,\widetilde{s_1},\cdots,\widetilde{s_l})$.

Since $\mathcal{S}$ is normal, there is a umers $\widehat{\Sigma}$ defining the set $\widehat{W}$. Now let be given an existential expression H^*. One effectively obtains a string $w^* \in \{r_0,r_1\}^+$ such that $H^* = H_{w^*}$. Say $w^* = r_{i_1}\cdots r_{i_{n^*}}$, with $n^* \in \mathbb{N}_+$, $i_1,\cdots,i_{n^*} \in \{0,1\}$.

Moreover, since $\mathcal{S}$ has a base constant and r_0, r_1 are constructible, there are ground terms t_0 and t_1 over $\mathcal{S}$, with the values r_0 and r_1, respectively.

Now we define the umeds Σ^* by

$$\begin{aligned}\Sigma^*(n,k) = H(&t_0,y_1,t_0,\cdots,t_0,y_n,t_1,t_{i_1},t_0,\cdots,t_0,t_{i_{n^*}})\\ &\text{if } \widehat{\Sigma}(\,2(n+n^*)\,,\,k\,) = H(x_1,x_2,\cdots,x_{2(n+n^*)}), \text{ for some expression } H.\end{aligned}$$

(If necessary, recall the definition of the pairing function *pair* from Section 4; notice that $2(n+n^*) = length(pair(s_1\cdots s_n, r_{i_1}\cdots r_{i_{n^*}}))$.)

Obviously, a (code of a) program computing Σ^* can effectively be obtained from H^* by a suitable modification of a (code of a) program which computes $\widehat{\Sigma}$.

So it remains to show that Σ^* defines W_{H^*}. This holds, since $s_1\cdots s_n \in W_{H^*}$ iff $\mathcal{S} \models H^*(s_1,\cdots,s_n)$ iff $pair(s_1\cdots s_n, w^*) \in \widehat{W}$ iff $pair(s_1\cdots s_n, w^*) \in W_{\widehat{\Sigma}(\,2(n+n^*)\,,\,k\,)}$, for some $k \in \mathbb{N}_+$, iff $s_1\cdots s_n \in W_{\Sigma^*(n,k)}$, for some $k \in \mathbb{N}_+$. □

Using the effective procedure described in the second part of the proof, one can even construct a procedure which, to every (code of a) program computing some umeers, yields a (code of a) program computing a umers defining the same set of strings. Thus, we have the following corollary, which also could directly be proved similar to the second part of the preceding proof.

Corollary 7.2 *A structure of finite signature with at least one base constant is normal iff there is an effective procedure which, for every (code of a classical) program computing a umeers, yields a (code of a classical) program that computes a umers defining the same set of strings.* □

Finally, we remark that also the results on um[e]eds and um[e]ers can straightforwardly be transferred to quasicomputability resp. quasirecognizability by allowing that quasiconstants occur within the [existentialized] conditional expressions. This simply follows by means of Lemma 3.3.

Analogously to the concept of normality, a structure can said to be *quasinormal* if every range of a D–quasicomputable function is also a domain of such a function. Lemma 3.3 implies that every normal structure is quasinormal, too. The following example shows that the converse does not hold.

Let $M \subseteq I\!N$ be a non–enumerable set (in the classical sense); χ_M denotes its characteristic function. We consider the structure

$$\mathcal{N}_a = \langle\, I\!N \cup \{a\}\,;\, 0\,;\, =\,;\, \nu\,,\, \widetilde{\chi_M}\,\rangle\,,$$

where $a \notin I\!N$ is a new element; $\nu(n) = n + 1$, for $n \in I\!N$, and $\nu(a) = a$; $\widetilde{\chi_M}(a, n) = \chi_M(n)$, for $n \in I\!N$, and $\widetilde{\chi_M}(x, y) = y$ if $x \neq a$ or $y = a$. The deterministically $\mathcal{N}_a$–recognizable subsets of $I\!N$ (whose elements have to be considered as strings of length 1) are just the classically enumerable sets of numbers. Indeed, starting with some input $n \in I\!N$, a deterministic $\mathcal{N}_a$–program $\mathcal{P}$ cannot generate the element a. Thus, there is a deterministic $\mathcal{N}$–program equivalent to $\mathcal{P}$ on such inputs.

On the other hand, there is an N_2–program which recognizes the set M over $\mathcal{N}_a$. Such a program can first guess the element a (which can be identified as the only element x satisfying the property "$\nu(x) = x$"), then the characteristic function of M is available. Similarly, M can also be represented as the output set of a deterministic $\mathcal{N}_a$–program.

With respect to quasicomputability over $\mathcal{N}_a$, however, the (deterministic) halting sets coincides with the output sets or the N_2–recognizable sets. This follows, since the universe, $I\!N \cup \{a\}$, is enumerated by the function ψ with

$$\psi(w) = \begin{cases} 0 & \text{if } w = a, \\ \nu(n) & \text{if } w = n \in I\!N, \\ w & \text{if } \mathit{length}(w) \geq 1, \end{cases}$$

and ψ is quasicomputable by means of the quasiconstant a.

8 Universal Programs etc., over Finite Signatures

Throughout this section, we suppose that the considered structures $\mathcal{S}$ are of finite signature.

Using the two constructible elements r_0 and r_1, every $\mathcal{S}$–quasiprogram $\mathcal{P}$ can be encoded in a straightforward manner by a string denoted by $code(\mathcal{P})$. To this purpose, let the finitely many *syntactical units* of programs, i.e., the keywords of our programming language, the technical symbols, base constants, base relations and base functions be encoded by strings from $\{r_0, r_1\}^+$, and indices of pointer variables and the goal labels be binarily encoded over $\{r_0, r_1\}$. The quasiconstants of quasiprograms are encoded by themselves.

Moreover, it is convenient to use only the track of even–numbered places in strings for these encodings, whereas the odd–numbered places are filled by r_0 or r_1, such that the starting places of codes of the syntactic units can uniquely be identified. More precisely, instead of the direct encoding "$s_1 s_2 \cdots s_n$" of some syntactic unit u, we use the padded string

$$code(u) = \text{ " } r_1\, s_1 r_0 s_2 \cdots r_0 s_n \text{ " } .$$

Then, for a quasiprogram $\mathcal{P} = u_1 u_2 \cdots u_m$, where the u_i are the syntactic units $(1 \leq i \leq m)$, let

$$code(\mathcal{P}) = code(u_1) \cdot code(u_2) \cdot \cdots \cdot code(u_m) .$$

Now, the parts of $code(\mathcal{P})$ which represent the codes of the syntactic units can be identified by a determinstic $\mathcal{S}$–program.

One straightforwardly shows

Lemma 8.1 *The sets of all codes of D–, N_1– and N_2–programs and of such quasiprograms, respectively, are D–decidable over the structure $\mathcal{S}$.* □

In the following, by $\varphi_{\mathcal{P}}$ and $\varrho_{\mathcal{P}}$, we denote the string function and the string relation, respectively, which is computed by the deterministic resp. nondeterministic program $\mathcal{P}$.

Theorem 8.1 *There is a D–program $\mathcal{U}$ such that, for all D–quasiprograms $\mathcal{P}$ and all strings $w \in S^+$, it holds*

$$\varphi_{\mathcal{U}}(\, pair(code(\mathcal{P}), w)\,) \cong \varphi_{\mathcal{P}}(\, w\,) .$$

For $i = 1, 2$, there is an N_i–program $\mathcal{U}_i$ such that, for all N_i–quasiprograms $\mathcal{P}$, it holds

$$\varrho_{\mathcal{P}} = \{(w, w') : (\, pair(code(\mathcal{P}), w)\,,\, w') \in \varrho_{\mathcal{U}_i}\} .$$

This is proved by applying standard techniques of simulation and programming, as they are well–known from classical computation theory based on the concept of Turing machine. The details are omitted here. Notice that our concept of $\mathcal{S}$–program corresponds to the notion of multihead Turing machine. Thus, the simulated program $\mathcal{P}$ may use arbitrarily many pointer variables, whereas the universal programs $\mathcal{U}$ and $\mathcal{U}_i$, respectively, are equipped with some fixed number of pointers only. Thus, in each step of the simulation, the positions of the $\mathcal{P}$–pointers p_j must be marked by the simulating program $\mathcal{U}_{[i]}$ by means of encodings of p_j at the corresponding places of the (encoding of the) current $\mathcal{P}$–configuration, cf. the proof of Proposition 4.6. To compute the values of base functions

or base relations, the universal programs can use (finally many) suitable subroutines. The simulation of nondeterministic steps can analogously be performed by subroutines. □

In the remaining part of this section, we restrict ourselves to deterministic programs and sketch how to obtain results analogous to some basic theorems of classical recursion theory. Whereas the case of quasicomputability can rather straightforwardly be treated (cf. also Section 9), in considering $\mathcal{S}$–computability one has to be more careful. So here we deal with the latter one.

The encoding of pairs of strings is used to define inductively the encoding of k–tuples of strings, for $k \in I\!N^+$. Let

$$tuple_1(w) = w, \quad tuple_2(w, w') = pair(w, w'), \quad \text{and}$$
$$tuple_{k+1}(w_0, w_1, \cdots, w_k) = pair(w_0, tuple_k(w_1, \cdots, w_k)), \quad \text{for } k > 1.$$

We simply write $[w_1, \cdots, w_k]$ instead of $tuple_k(w_1, \cdots, w_k)$, for $k \geq 1$.

A k–ary partial function $\varphi : (S^+)^k \succ\!\!\!\rightarrow S^+$ is said to be (deterministically) *computable* over $\mathcal{S}$ iff the unary string function $\bar{\varphi}$ is deterministically $\mathcal{S}$–computable, where

$$\bar{\varphi}([w_1, \cdots, w_k]) \cong \varphi(w_1, \cdots, w_k), \text{ for } w_1, \cdots, w_k \in S^+, \quad \text{and}$$
$$\bar{\varphi}(w) \text{ is undefined, for } w \notin \{[w_1, \cdots, w_k] : w_1, \cdots, w_k \in S^+\}.$$

Finally, we write $\Phi_w(w')$ instead of $\varphi_{\mathcal{U}}(\, pair(w, w')\,)$, with respect to some fixed universal program $\mathcal{U}$ according to Theorem 8.1. Without loss of generality, we suppose that $\varphi_{\mathcal{U}}(\, pair(w, w')\,)$ is undefined if $w \notin \{r_0, r_1\}^+$.

Thus, for every $w \in S^+$, Φ_w denotes a deterministically $\mathcal{S}$–computable partial string function, and w can be considered to be the code of a corresponding D–program. Conversely, by Theorem 8.1, to every deterministically $\mathcal{S}$–computable unary partial function $\varphi : S^+ \succ\!\!\!\rightarrow S^+$, there is a string $w \in \{r_0, r_1\}^+$ such that $\varphi = \Phi_w$.

We have the following s-m-n theorem.

Proposition 8.1 (s-m-n Theorem) *To every $m, n \in I\!N_+$, there is a deterministically $\mathcal{S}$–computable $(m+1)$-ary total function $\sigma_n^m : (\{r_0, r_1\}^+)^{m+1} \longrightarrow \{r_0, r_1\}^+$ such that*

$$\Phi_{w_0}([w_1, \cdots, w_m, w_{m+1}, \cdots, w_{m+n}]) \cong \Phi_{\sigma_n^m(w_0, w_1, \cdots, w_m)}([w_{m+1}, \cdots, w_{m+n}]),$$

for all $w_0, w_1, \cdots, w_m \in \{r_0, r_1\}^+$ and all $w_{m+1}, \cdots, w_{m+n} \in S^+$.

Indeed, for given $w_0, w_1, \cdots, w_m$, the string $\Phi_{w_0}([w_1, \cdots, w_m, w_{m+1}, \cdots, w_{m+n}])$ can be computed from the input $[w_{m+1}, \cdots, w_{m+n}]$ by a program $\mathcal{P}_{(w_0, w_1, \cdots, w_m)}$ which transforms the input into $[w_1, \cdots, w_m, w_{m+1}, \cdots, w_{m+n}]$, in a preprocessing stage, and works according to (the universal program computing) Φ_{w_0} then. Now, as in classical theory of computation, there is a program which transfers all codes of tuples $[w_0, w_1, \cdots, w_m]$ into codes of such programs $\mathcal{P}_{(w_0, w_1, \cdots, w_m)}$ by modifying the code of the universal program $\mathcal{U}$. □

We remark that the restriction to strings $w_0, w_1, \cdots, w_m \in \{r_0, r_1\}^+$ is essential if we want to deal with proper programs only. For $w_1, \cdots, w_m \in S^+$, $\mathcal{P}_{(w_0, w_1, \cdots, w_m)}$ is only a quasiprogram in general.

Now one obtains the recursion theorem, the fixed–point theorem and Rice's theorem in the same way as in the classical theory, cf. [16, 56].

Proposition 8.2 (Recursion Theorem) *Let $n \in I\!N_+$, and $\varphi : (S^+)^{n+1} \succ\!\!\rightarrow S^+$ be a deterministically $\mathcal{S}$–computable function. There is a string $w_0 \in \{r_0, r_1\}^+$ such that, for all $w_1, \cdots, w_n \in S^+$,*

$$\varphi(w_0, w_1, \cdots, w_n) \cong \Phi_{w_0}([w_1, \cdots, w_n]).$$

To show this, we define

$$\psi(u, w_1, \cdots, w_n) \cong \varphi(\sigma_n^1(u, u), w_1, \cdots, w_n).$$

Since ψ is an $\mathcal{S}$–computable function, there exits a string $w \in \{r_0, r_1\}^+$ such that

$$\psi(u, w_1, \cdots, w_n) \cong \Phi_w([u, w_1, \cdots, w_n]) \cong \Phi_{\sigma_n^1(w,u)}([w_1, \cdots, w_n]).$$

For $u = w$, and $w_0 = \sigma_n^1(w, w)$, we have $\varphi(w_0, w_1, \cdots, w_n) \cong \Phi_{w_0}([w_1, \cdots, w_n])$. □

Proposition 8.3 (Fixed–Point Theorem) *Let $n \in I\!N_+$, $\varphi : S^+ \succ\!\!\rightarrow S^+$ be a unary deterministically $\mathcal{S}$–computable function. There is a string $w_0 \in \{r_0, r_1\}^+$ such that, for all $w_1, \cdots, w_n \in S^+$,*

$$\Phi_{w_0}([w_1, \cdots, w_n]) \cong \Phi_{\varphi(w_0)}([w_1, \cdots, w_n]).$$

This is proved by considering the function

$$\psi(w, w_1, \cdots, w_n) \cong \Phi_{\varphi(w)}([w_1, \cdots, w_n]).$$

By the recursion theorem, there is a string $w_0 \in \{r_0, r_1\}^+$ such that

$$\psi(w_0, w_1, \cdots, w_n) \cong \Phi_{w_0}([w_1, \cdots, w_n]). \square$$

Proposition 8.4 (Rice's Theorem) *Let $\mathcal{F}$ be a set of deterministically $\mathcal{S}$–computable unary partial functions which does not contain all such functions but does contain the empty function $\emptyset$. Then the index set*

$$\mathcal{I}(\mathcal{F}) = \{w : w \in \{r_0, r_1\}^+, \Phi_w \in \mathcal{F}\}$$

is not (deterministically) $\mathcal{S}$–recognizable.

If $\mathcal{I}(\mathcal{F})$ would be recognizable, the function

$$\varphi(w) = \begin{cases} w_- & \text{if } w \in \mathcal{I}(\mathcal{F}), \\ \text{undefined} & \text{otherwise,} \end{cases}$$

where w_- is an index of a deterministically $\mathcal{S}$–computable unary partial function which does not belong to $\mathcal{F}$, would be computable. Thus, by the fixed–point theorem, there is an index $w_0 \in \{r_0, r_1\}^+$ such that, for all $w \in S^+$,

$$\Phi_{w_0}(w) \cong \Phi_{\varphi(w_0)}(w).$$

From $w_0 \in \mathcal{I}(\mathcal{F})$, it would follow $\varphi(w_0) = w_- \notin \mathcal{I}(\mathcal{F})$, in contrast to the above equation.

Thus, we have $w_0 \notin \mathcal{I}(\mathcal{F})$, and $\varphi(w_0)$ is undefined. Then Φ_{w_0} must be the empty function $\emptyset \in \mathcal{F}$, i.e. $w_0 \in \mathcal{I}(\mathcal{F})$. This is a contradiction, too. □

From Propositions 8.4 and 4.1, it follows

Corollary 8.1 *Under the suppositions of Proposition 8.4, the index set $\mathcal{I}(\mathcal{F})$ is not recursively enumerable in the classical sense.* □

9 Effectively Encodable Structures

Generalized computability theory has usually been studied with respect to structures of finite signatures. As we have tried to demonstrate in Sections 2–7, however, our approach works also on structures of infinite signatures. Of course, the programs are not necessarily representable as strings over some finite alphabet then, but this is already not the case for quasiprograms over finite signatures.

Now we are going to deal with universality and related concepts over arbitrary signatures. These investigations are especially encouraged by S. Smale's question if there are NP–complete sets over the structures $\mathcal{R}_{lin}$ or $\mathcal{R}_{lin,\leq}$, see [43, 44, 47]. K. Meer [43] conjectured that this is not the case, since these structures don't even own universal functions. Recently, C. Gaßner [25] has shown that both structures have NP–complete sets. So it seems to be appropriate to consider the concept of universal function a little more detailed. Moreover, in the next section, we shall deal with m–completeness which is the recursive analogue of NP–completeness.

We consider bipotent structures

$$\mathcal{S} = \langle\ S\ ;\ (C_i : i \in I_C)\,;\ (R_i : i \in I_R)\,;\ (F_i : i \in I_F)\ \rangle\,,$$

with arbitrary index sets I_C, I_R, I_F, and constructible elements $r_0, r_1 \in S$.

By an *effective encoding* of $\mathcal{S}$, we understand a triple

$$\kappa = \langle\ \kappa_C\ ,\ \kappa_R\ ,\ \kappa_F\ \rangle$$

of surjective functions $\kappa_C : S^+ \succ\!\!\!\rightarrow I_C$, $\kappa_R : S^+ \succ\!\!\!\rightarrow I_R$, $\kappa_F : S^+ \succ\!\!\!\rightarrow I_F$, such that there are $\mathcal{S}$–computable string functions φ_C, φ_R, φ_F satisfying

$$\begin{aligned}
\varphi_C(w) &= C_{\kappa_C(w)}, && \text{for all } w \in dom(\kappa_C),\\
\varphi_R(\, pair(w, s_1 \cdots s_{k_i})\,) &= \chi^{r_0,r_1}_{R_{\kappa_R(w)}}(s_1, \cdots, s_{k_i}), && \text{for all } w \in dom(\kappa_R),\ s_1, \cdots, s_{k_i} \in S,\\
\varphi_F(\, pair(w, s_1 \cdots s_{l_i})\,) &= F_{\kappa_F(w)}(s_1, \cdots, s_{k_i}), && \text{for all } w \in dom(\kappa_F),\ s_1, \cdots, s_{l_i} \in S.
\end{aligned}$$

Here, $\chi^{r_0,r_1}_{R_{\kappa_R(w)}}$ denotes the characteristic function of the relation $R_{\kappa_R(w)}$, with values from $\{r_0, r_1\}$; more precisely,

$$\chi^{r_0,r_1}_{R_{\kappa_R(w)}}(s_1, \cdots, s_{k_i}) = \begin{cases} r_1 & \text{if } (s_1, \cdots, s_{k_i}) \in R_{\kappa_R(w)},\\ r_0 & \text{if } (s_1, \cdots, s_{k_i}) \notin R_{\kappa_R(w)}. \end{cases}$$

Roughtly speaking, the definition means that the base constants, relations and functions are encoded by strings of elements of the structure such that the constants can $\mathcal{S}$–effectively be obtained from their codes, and, given its code and a string (of appropriate length) of arguments, the value of any base relation resp. function can $\mathcal{S}$–effectively be computed, too. There is no other requirement of effectivity than that of the computability of the functions φ_C, φ_R and φ_F. In particular, the sets of all codes of base constants, relations and functions, $\kappa_C^{-1}(I_C), \kappa_R^{-1}(I_R), \kappa_F^{-1}(I_F)$, are not required to be effective in any sense (p.e., recognizable). Thus, for any effective encoding κ like above, every triple $\kappa' = \langle \kappa'_C, \kappa'_R, \kappa'_F \rangle$, where $\kappa'_C \subseteq \kappa_C, \kappa'_R \subseteq \kappa_R, \kappa'_F \subseteq \kappa_F$, is an effective encoding too, as long as $\kappa'_C, \kappa'_R, \kappa'_F$ are mappings onto the corresponding index sets. This means, an effective encoding can always be assumed to consist of biunique functions $\kappa_C, \kappa_R, \kappa_F$.

From the uniqueness of the encodings, it immediately follows that effectively encodable structures can have at most $card(S^+)$ many base constants, relations and functions.

Lemma 9.1 *If both I_R and I_F are finite, the structure $\mathcal{S}$ is effectively encodable.*

For the proof, we assume without loss of generality that the index sets of the families of base relations and base functions consist of positive natural numbers, $I_R, I_F \subseteq I\!N_+$. Then the base relations resp. functions are encoded by the 'binary' representations (within the alphabet $\{r_0, r_1\}$) of their indices. The base constants are encoded by themselves (each considered as a string of length 1). Therefore, there is no restriction of their cardinality. It is simple to show the existence of functions φ_C, φ_R, φ_F, according to the definition. □

The just constructed effective encodings for structures with finitely many base relation and functions will be denoted as *standard encodings* in the sequel.

The notion of universal function is straightforwardly defined: a partial string function $\varphi_u : S^+ \succ\!\!\rightarrow S^+$ is said to be *universal* (with respect to deterministic computability) if it is deterministically $\mathcal{S}$–computable and, for every $\mathcal{S}$–computable function $\psi : S^+ \succ\!\!\rightarrow S^+$, there exists a string $w_\psi \in S^+$ such that

$$\varphi_u(\, pair(w_\psi, v)\,) \;\cong\; \psi(v)\,, \text{ for all } v \in S^+\,.$$

Theorem 9.1 *A structure is effectively encodable iff it owns a universal function.*

If there is a universal function φ_u, one can define an effective encoding in the following way. For every base constant C_i, the constant function $\psi_{C,i}$, with $\psi_{C,i}(w) = C_i$, for all $w \in S^+$, is $\mathcal{S}$–computable. Thus, the strings $w_{C,i}$, for which $\varphi_u(\, pair(w_{\psi_{C,i}}, v)\,) = C_i$, for all $v \in S^+$, can be taken as codes of the constants C_i ($i \in I_C$). The function φ_C defined by

$$\varphi_C(w) \;\cong\; \varphi_u(\, pair(w, r_0)\,),\;\; w \in S^+,$$

satisfies the requirement from the definition of effective encodings.

For every base relation R_i, the function $\psi_{R,i}$ defined by

$$\psi_{R,i}(w) = \begin{cases} r_1 & \text{if } w = s_1 \cdots s_{k_i} \text{ and } (s_1, \cdots, s_{k_i}) \in R_i, \\ r_0 & \text{if } w = s_1 \cdots s_{k_i} \text{ and } (s_1, \cdots, s_{k_i}) \notin R_i, \\ \text{undefined} & \text{otherwise (i.e., } length(w) \neq k_i), \end{cases}$$

is deterministically $\mathcal{S}$–computable. Then the string $w_{R,i}$, with

$$\varphi_u(\, pair(w_{R,i}\,, v)\,) \;\cong\; \psi_{R,i}(v)\,, \text{ for all } v \in S^+\,,$$

can be taken as a code of relation R_i ($i \in I_R$). $\varphi_R = \varphi_u$ satisfies the related requirement.

The codes of the base functions are analogously defined.

To prove the converse direction of the assertion, let $\kappa = \langle\, \kappa_C\,,\, \kappa_R\,,\, \kappa_F\,\rangle$ be an effective encoding of a structure $\mathcal{S}$. Now the code $code_\kappa(\mathcal{P})$ of an $\mathcal{S}$–quasiprogram $\mathcal{P}$ can be defined quite analogously to the case of finite signature in Section 8, only that the base constants, relations and functions occurring in $\mathcal{P}$ are represented by means of their κ–codes instead of strings from $\{r_0, r_1\}^+$. (For the standard encoding of a structure of finite signature, we

obtain the code as used in Section 8.) In order to obtain a unique code of each quasiprogram, we may suppose that the encodings $\kappa_C, \kappa_R, \kappa_F$ are biunique.

Analogously to Theorem 8.1, there is a universal program $\mathcal{U}_\kappa$ which, for a given pair of a code of some quasiprogram $\mathcal{P}$ and an input string $w \in S^+$, simulates the behaviour of $\mathcal{P}$ on that input string w. Three programs computing the functions φ_C, φ_R and φ_F, respectively, have to serve as subroutines in the course of these simulations. $\Box$

Notice that, also analogously to Theorem 8.1, there are nondeterministic universal programs, for both kinds of nondeterminism, too. In contrast to Lemma 8.1 however, the sets of all codes of (D–, N_1– resp. N_2–) programs or quasiprograms are not necessarily decidable.

It has turned out that, on any effectively encodable structure, we even obtain a properly computable function which is universal for all *quasi*computable functions. It is interesting to notice that, in contrast to the remark on normality at the end of Section 7, it does not yield a new concept if we consider quasi–effective encodings or quasi–universal functions in the sense that the functions $\varphi_C, \varphi_R, \varphi_F$ and φ_u, respectively, have to be quasicomputable only. This follows, since the quasiconstants which may be used to compute those functions can alternatively be given by prefixes of the code strings, whereas the correspondingly modified functions $\varphi'_C, \varphi'_R, \varphi'_F$ and φ'_u, respectively, become properly computable, cf. Lemma 3.3.

Now we show that the basic ingredients of recursion theory rather straightforwardly hold over arbitrary effectively encodable structures.

Theorem 9.2 *For every effectively encodable structure $\mathcal{S}$, there is a D–computable function φ_u which is universal for the set of all D–quasicomputable functions over $\mathcal{S}$ and satisfies the corresponding s-m-n theorem, recursion theorem, fixed–point theorem, and Rice's theorem. More precisely, if we define*

$$\Phi_w(v) \cong \varphi_u(\, pair(w, v)\,), \text{ for all } w, v \in S^+,$$

every Φ_w represents a D–quasicomputable function, and, for every D–quasicomputable string function ψ, there is an index w_ψ such that $\Phi_{w_\psi}(\, v\,) \cong \psi(\, v\,)$, for all $v \in S^+$. Moreover, we have:

1. **s-m-n Theorem**. *To every $m, n \in \mathbb{N}_+$, there is a D–computable $(m+1)$-ary total function $\sigma_n^m : (S^+)^{m+1} \longrightarrow S^+$ such that, for all $w_0, \cdots, w_m, \cdots, w_{m+n} \in S^+$,*

$$\Phi_{w_0}([w_1, \cdots, w_m, w_{m+1}, \cdots, w_{m+n}]) \cong \Phi_{\sigma_n^m(w_0, w_1, \cdots, w_m)}([w_{m+1}, \cdots, w_{m+n}]).$$

2. **Recursion Theorem**. *Let $n \in \mathbb{N}_+$, $\varphi : (S^+)^{n+1} \succ\!\!\!\rightarrow S^+$ be a D–quasicomputable function. There is a string $w_0 \in S^+$ such that, for all $w_1, \cdots, w_n \in S^+$,*

$$\varphi(w_0, w_1, \cdots, w_n) \cong \Phi_{w_0}([w_1, \cdots, w_n]).$$

3. **Fixed–Point Theorem**. *Let $n \in \mathbb{N}_+$, $\varphi : S^+ \succ\!\!\!\rightarrow S^+$ be a D–quasicomputable unary function. There is a string $w_0 \in S^+$ such that, for all $w_1, \cdots, w_n \in S^+$,*

$$\Phi_{w_0}([w_1, \cdots, w_n]) \cong \Phi_{\varphi(w_0)}([w_1, \cdots, w_n]).$$

4. **Rice's Theorem**. *Let $\mathcal{F}$ be a set of D–quasicomputable unary functions which contains the empty function $\emptyset$, but does not contain all D–quasicomputable functions. Then the index set*

$$\mathcal{I}(\mathcal{F}) = \{w : w \in S^+ \,,\, \Phi_w \in \mathcal{F}\}$$

is not $\mathcal{S}$–quasirecognizable.

If the codes of quasiprograms and the universal function φ_u are defined like described in the sketch of proof of Theorem 9.1, the assertions of the present theorem can be proved quite similarly to their analogues from Section 8. □

Proposition 9.1 *The structures $\mathcal{R}_{lin}$, $\mathcal{R}_{lin,\leq}$, $\mathcal{R}_{sc}$ are not effectively encodable.*

Assume, there is a universal function φ_u over $\mathcal{R}_{lin}$. Let $r_1, \cdots, r_l$ denote the scalar factors of multiplications used by some $\mathcal{R}_{lin}$–program $\mathcal{P}_u$ computing φ_u.

For every $r \in I\!R$, there must be a string $w_r \in I\!R^+$ such that

$$\varphi_u(\, pair(w_r, x)\,) = r \cdot x \,, \text{ for every } x \in I\!R.$$

Let $w_r = s_1 \cdots s_{n_r}$, where $s_1, \cdots, s_{n_r} \in I\!R$. Then the elements y occurring in $\mathcal{P}_u$–computations on inputs $x \in I\!R$, can be represented in the form

$$y = \sum_{j=1}^{m} r_1^{\alpha_{j1}} \cdot \ldots \cdot r_l^{\alpha_{jl}} \cdot z_j \;+\; \sum_{j=1}^{m'} r_1^{\alpha'_{j1}} \cdot \ldots \cdot r_l^{\alpha'_{jl}} \,,$$

with $m, m' \in I\!N$, $\alpha_{j1}, \cdots, \alpha_{jl}, \alpha'_{j1}, \cdots, \alpha'_{jl} \in I\!N$, and $z_j \in \{x, s_1, \cdots, s_{n_r}\}$. (Notice that the dots " $\cdot$ " denote multiplications in the representation of y.)

Thus, y can take countably many values which don't depend on the input x (for the cases where $z_j \neq x$, for all $j \in \{1, \cdots, m\}$, in the representation of y), or values which belongs to countably many straight lines in the (x, y)–plane whose slopes are from the set

$$A_u \;=\; \{\, \sum_{j=1}^{m} r_1^{\alpha_{j1}} \cdot \ldots \cdot r_l^{\alpha_{jl}} \;:\; m \in I\!N,\, \alpha_{j\lambda} \in I\!N \,(1 \leq j \leq m,\, 1 \leq \lambda \leq l)\,\}.$$

Therefore, if we choose some $r \in I\!R \setminus A_u$, then, on inputs $x \in I\!R$, the program $\mathcal{P}_u$ yields only countably many values. On the other hand, we have $\{\varphi_u(w_r, x) : x \in I\!R\} = I\!R$. Thus, the assumption is wrong.

The proofs for the structures $\mathcal{R}_{lin,\leq}$ and $\mathcal{R}_{sc}$ are analogous. □

One also shows easily that the structure $\mathcal{N}_{sc} = \langle I\!N_+ \,; 1; =; (\mu_p \,:\, p \in \mathit{Prim})\rangle$, where *Prim* denotes the set of prime numbers, is not effectively encodable. On the other hand, the linear structure over the natural numbers, $\mathcal{N}_{lin} = \langle I\!N_+ \,; 1; =; (\mu_p \,:\, p \in \mathit{Prim}), +\rangle$, is computationally equivalent to the standard structure of natural numbers, $\mathcal{N}$, which is of finite signature. Thus, there is a universal function over $\mathcal{N}_{lin}$.

Now we are going to descibe two classes of effectively encodable structures.

First, let

$$\mathcal{S} = \langle\; S \;;\; \mathbf{C} \,;\, \mathbf{R} \,;\, \mathbf{O_2} \;\rangle$$

be a structure whose family of functions, $\mathbf{O_2} = (F_i : i \in I_F)$, consists of binary operations only. We consider the family of all unary operations obtained from $\mathbf{O_2}$ by fixing the first arguments:

$$\mathbf{O_1} = (F_{(i,s')} : i \in I_F,\ s' \in S),$$

where $F_{(i,s')}(x) = F_i(s', x)$, for all $x \in S$.

The *Megiddo extension* of $\mathcal{S}$ is defined by adding all these unary functions to the family of operations,

$$EX_{Meg}(\mathcal{S}) = \langle\ S\ ;\ \mathbf{C}\ ;\ \mathbf{R}\ ;\ \mathbf{O_1}\ ,\ \mathbf{O_2}\ \rangle.$$

Lemma 9.2 *If $\mathcal{S}$ is effectively encodable, then $EX_{Meg}(\mathcal{S})$ too.*

Indeed, the additional operations $F_{(i,s')}$ can be encoded by the strings $pair(w_i, s')$, where w_i is a code of the base function F_i ($i \in I_F$). □

N. Megiddo [47] considered such extensions (of structures $\mathcal{S}$ of finite signatures) in order to show that his basic construction of NP–complete sets also applies to certain structures of infinite signatures. This was encouraged by Smale's question if there are NP–complete sets over $\mathcal{R}_{lin}$ and/or $\mathcal{R}_{lin,\leq}$, cf. our remarks at the beginning of this section. By Proposition 9.1 and Lemma 9.3, it is obvious that the linear structures over the reals are not Megiddo extensions of any effectively encodable structures.

A second type of effectively encodable structures is obtained by adding all constructible elements, all decidable relations, and all computable total functions over the universe to some structure's base. More precisely, the *computational extension* of a structure $\mathcal{S}$ is

$$EX_{Comp}(\mathcal{S}) = \langle\ S\ ;\ \mathbf{C}_{Comp}\ ;\ \mathbf{R}_{Comp}\ ;\ \mathbf{F}_{Comp}\ \rangle,$$

where
$\mathbf{C}_{Comp}$ $= (s : s$ is an $\mathcal{S}$–constructible element),
$\mathbf{R}_{Comp}$ $= (r : r \subseteq S^k$ for some $k \in \mathbb{N}_+$, r is $\mathcal{S}$–decidable),
$\mathbf{F}_{Comp}$ $= (f : f : S^l \longrightarrow S$ for some $l \in \mathbb{N}_+$, and f is $\mathcal{S}$–computable).

The decidability of relations r and the computability of functions f as they occur in the definition have to be understood straightforwardly.

Theorem 9.2 implies that EX_{Comp} is effectively encodable if $\mathcal{S}$ is. Indeed, the constructible elements are encoded by themselves, and as the function φ_C one takes the identity. The relations and functions of the computational extension can be encoded by means of codes of $\mathcal{S}$–programs computing them, where their arity has to be marked additionally. The functions φ_R and φ_F are obtained from a universal function for the structure $\mathcal{S}$.

Now it is obvious that, by the same method, also the *quasicomputable extension* is shown to be effectively encodable. This is the structure

$$EX_{QComp}(\mathcal{S}) = \langle\ S\ ;\ \mathbf{C}_{QComp}\ ;\ \mathbf{R}_{QComp}\ ;\ \mathbf{F}_{QComp}\ \rangle,$$

where $\mathbf{C}_{QCom} = S$, $\mathbf{R}_{QComp}$ consists of all quasidecidable relations, and $\mathbf{F}_{QComp}$ consists of all quasicomputable total functions (both in $\mathcal{S}$, with arbitrary arities).

Lemma 9.3 *If the structure $\mathcal{S}$ is effectively encodable, then EX_{Comp} and EX_{QComp}, too.* □

The [quasi]computational extensions of structures are interesting, since they characterize the [quasi]computational power of the original structures. Two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ with the same universe S are said to be *[quasi]computationally equivalent* if their classes of [quasi]computable string functions (over S^+) coincide. For example, one sees easily

Lemma 9.4 *Two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ are [quasi]computationally equivalent iff their [quasi]computational extensions coincide, i.e., $EX_{[Q]Comp}(\mathcal{S}_1) = EX_{[Q]Comp}(\mathcal{S}_2)$.* □

10 m–Completeness

Analogously to classical recursion theory, a set of strings, $W \subseteq S^+$, is said to be m–*complete* if it is recognizable (in the basic sense of Section 2) and, moreover, every other recognizable set $V \subseteq S^+$ is m–*reducible* to W.
The latter means that there is a (D–) computable total string function $\varrho : S^+ \longrightarrow S^+$ such that

$$v \in V \text{ iff } \varrho(v) \in W, \quad \text{for all } v \in S^+.$$

The function ϱ is also called an m–*reduction* in this case, and we write briefly: $V \leq_m W$.

If the structure is effectively encodable and φ_u a universal function, the *general halting problem* (with respect to φ_u) is the set

$$\mathrm{GH} = \{\, [w,v] : \varphi_u([w,v]) \text{ is defined} \,\},$$

where "$[w,v]$" abbreviates "$pair(w,v)$", cf. Section 8.

Lemma 10.1 *For an effectively encodable structure, GH is recognizable but not decidable.*

The first assertion holds by Proposition 6.1: $\mathrm{GH} = dom(\varphi_u) \cap \{[w,v] : w,v \in S^+\}$, and the set of all (codes of) pairs is decidable.

The second assertion follows by the standard diagonalization: if GH would be decidable, there would exist a computable function ψ such that $\psi(v)$ is defined iff $[v,v] \notin \mathrm{GH}$ (for all $v \in S^+$). For an index w_ψ of ψ with respect to the universal function φ_u, we would obtain: $\varphi_u([w_\psi, w_\psi]) \cong \psi(w_\psi)$ is defined iff $[w_\psi, w_\psi] \notin \mathrm{GH}$ iff $\varphi_u([w_\psi, w_\psi])$ is not defined. □

More precisely, we have just shown that the set of all self–applicable codes,

$$K = \{w : \varphi_u([w,w]) \text{ is defined} \}$$

is not decidable, and that $K \leq_m \mathrm{GH}$. Thus, since the halting sets are closed under pre–images of computable functions, cf. Section 6, if GH would be decidable, K would be too.

Like in classical theory, one shows that $\mathrm{GH} \leq_m K$.

GH represents the classical prototype of an m–complete set. On the first view, this may seem to be true also within our setting. Indeed, a recognizable set $V = dom(\psi)$ seems to be m–reduced to GH by the function ϱ defined by

$$\varrho(v) = \varphi_u(\,[w,v]\,) \quad (v \in S^+),$$

where $w = w_\psi$ is an index of the function ψ. Unfortunately, to compute this function ϱ, one needs the constructibility of the string w.

We shall say that the universal function φ_u *admits constructible programs* if, for every $\mathcal{S}$–computable function ψ, there is an $\mathcal{S}$–constructible string $w_\psi \in S^+$ such that $\psi(v) \cong \varphi_u([w_\psi, v])$, for all $v \in S^+$. So, we have shown

Proposition 10.1 *If the structure owns a universal function which admits constructible programs, then the corresponding general halting problem is an m–complete set.* □

Analogously to Theorem 9.1, one can show that there is a universal function admitting constructible programs iff the structure is effectively encodable in such a way that all code strings (of base constants, base relations and base functions) are constructible. In particular, this holds for all structures of finite signatures.

The following theorem shows that the properties of structures to be effectively encodable and to own m–complete sets, respectively, are independent over general structures.

Theorem 10.1 *There are both effectively encodable structures that don't own m–complete sets and structures that own m–complete sets but are not effectively encodable.*

A bipotent structure which is not effectively encodable but owns an m–complete set is given by

$$\widetilde{\mathcal{N}} \;=\; \langle\, I\!N \,;\, 0, 1 \,;\, \leq \,;\, (\widetilde{\mu_n} : n \in I\!N_+ \,)\,\rangle\,,$$

where $\widetilde{\mu_n} : I\!N^2 \longrightarrow I\!N$ is defined as follows.

$$\widetilde{\mu_n}(x,y) = \begin{cases} n \cdot x & \text{if } n \cdot x < y, \\ x & \text{otherwise.} \end{cases}$$

Without loss of generality, let $r_0 = 0$ and $r_1 = 1$.

We first show that $\widetilde{\mathcal{N}}$ is not effectively encodable. Assume, there is a universal function φ_u over $\widetilde{\mathcal{N}}$. Then, for every $m \in I\!N$, there is a string w_m such that

$$\widetilde{\mu_m}(x,y) = \varphi(\,[w_m, xy]\,), \quad \text{for all } x, y \in S.$$

Let φ_u be computed by a program $\mathcal{P}_u$ which uses only the functions $\widetilde{\mu_{n_1}}, \cdots, \widetilde{\mu_{n_k}}$, for numbers $n_1, \cdots, n_k \in I\!N_+$. Thus, at every step of working of $\mathcal{P}_u$ on input strings $[w_m, xy]$, where $x, y \in I\!N$, all elements of the current string can be represented in the form

$$n_1^{\alpha_1} \cdot \ldots \cdot n_k^{\alpha_k} \cdot z\,,$$

where $z \in \{x, y, 0, 1\}$, or z is an element of the string w_m. (The dot " $\cdot$ " denotes the multiplication in this representation !)

Now let m have no prime divisor common to some of the numbers $n_1, \cdots, n_k$; and let p be a prime number which is neither a divisor of $m, n_1, \cdots, n_k$ nor of elements from w_m. If $w_m = s_1 \cdots s_{l_m}$, we consider

$$x \;=\; p \cdot \prod_{\lambda=1}^{l_m} s_\lambda\,,$$

and take a prime y greater than $m \cdot x$ as the second element of the argument string. In particular, y is not a divisor of elements of w_m.

Then

$$\widetilde{\mu_m}(x,y) \;=\; m \cdot x\,,$$

but we have seen that this product cannot be obtained as an element in the course of the computation of program $\mathcal{P}_u$ on the input string $[w_n, xy]$.

Therefore, $\widetilde{\mathcal{N}}$ cannot own a universal function.

Now we show how to construct an m–complete set over $\widetilde{\mathcal{N}}$. Here we follow the idea used by C. Gaßner to construct N_1P–complete sets for the linear structures over the reals. We sketch the main steps of our construction. For more details, the reader is referred to [25].

Let $\mathcal{P}$ be an $\widetilde{\mathcal{N}}$–program using only the functions $\widetilde{\mu_{n_1}}, \cdots, \widetilde{\mu_{n_l}}$ $(n_1, \cdots, n_l \in I\!N_+)$. The string *precode*$(\mathcal{P})$ is defined similar to the code of the program, cf. the beginning of Section 8. The difference is that, instead of the codes of the occurring functions, only their indices from $\{1, \cdots, l\}$ are noted (binarily encoded) at the corresponding places. Of course, this precode is not sufficient to simulate the behaviour of $\mathcal{P}$ on some input. For an input string $w = s_1 \cdots s_k$ $(s_1, \cdots, s_k \in I\!N)$, let the string *prod*$(\mathcal{P}, w)$ consist of all products of the form

$$n_1^{\alpha_1} \cdot \ldots \cdot n_l^{\alpha_l} \cdot s_\kappa \ ,$$

for all elements s_κ of w and all exponents $\alpha_\lambda \leq max(s_1, \cdots, s_k)$, where these products are arranged according to some predefined ordering, say lexicographically with respect to the vector $(\alpha_1, \cdots, \alpha_l, \kappa)$.

Given (the code of) a triple $[precode(\mathcal{P}), w, prod(\mathcal{P}, w)]$, a suitable $\widetilde{\mathcal{N}}$–program $\mathcal{P}_{pu}$ can simulate the behaviour of the program $\mathcal{P}$ on the input string w by taking the elements to be obtained in the course of that $\mathcal{P}$–computation from the string *prod*$(\mathcal{P}, w)$, by means of a suitable index processing. In particular, it stops if $\mathcal{P}$ would stop on the input string w. Such a $\mathcal{P}_{pu}$ could be called a *pre–universal* program over $\widetilde{\mathcal{N}}$. Now the reason for taking $\widetilde{\mu_n}$ (instead of the scalar multiplication μ_n) is obvious. In this case, for every input string $w = s_1 \cdots s_k$, the results of the multiplications are bounded by $max(s_1, \cdots, s_k)$, and a finite string of products is sufficient for the correct simulation.

Let

$$\begin{aligned} \widetilde{W} = \{\, [precode(\mathcal{P}), w, v] :\ & \mathcal{P} \text{ is an } \widetilde{\mathcal{N}}\text{–program}, \\ & w, v \in I\!N^+, \text{ and the pre–universal program } \mathcal{P}_{pu} \\ & \text{stops on the input string } [precode(\mathcal{P}), w, v] \,\}. \end{aligned}$$

Since the set of all precodes of $\widetilde{\mathcal{N}}$–programs is decidable, the set $\widetilde{W}$ is recognizable. Notice that the component v is not supposed to be the product string, $v = prod(\mathcal{P}, w)$. This property is not $\widetilde{\mathcal{N}}$–decidable.

Finally, we see that every $\widetilde{\mathcal{N}}$–recognizable set V is m–reducible to $\widetilde{W}$. Indeed, if V is the domain of some string function computed by an $\widetilde{\mathcal{N}}$–program $\mathcal{P}$, the function ϱ defined by

$$\varrho(w) = [precode(\mathcal{P}), w, prod(\mathcal{P}, w)]$$

is an m–reduction of V to $\widetilde{W}$.

We define an effectively encodable structure without m–complete sets as follows.

$$\widetilde{\mathcal{H}} \ = \ \langle\, I\!N^+ \cup \{e, e'\}\, ;\, e, e'\, ;\, (\alpha_n :\ n \in I\!N), * \,\rangle\, .$$

Let us stress that $I\!N^+$ is the set of all non–empty strings of natural numbers. "$*$" denotes the concatenation of strings, where e and e' are neutral elements, i.e.,

$$e * w = w * e = e' * w = w * e' = w\ , \qquad \text{for all } w \in I\!N^+.$$

Moreover, for $n \in I\!N$, let

$$\alpha_n(w) = \begin{cases} w * n & \text{if } w \in I\!N^+, \\ w & \text{if } w \in \{e, e'\}. \end{cases}$$

e and e' are the only $\widetilde{\mathcal{H}}$–constructible elements. They are used in order to obtain a bipotent structure.

The extension of $\widetilde{\mathcal{H}}$, $\widehat{\mathcal{H}} = \langle I\!N^+ \cup \{e,e'\}\,;\, e,e'\,;\, (\alpha_n : n \in I\!N \cup \{e,e'\}), * \rangle$, where both α_e and $\alpha_{e'}$ is the identity, is just the Megiddo extension of the structure $\langle I\!N^+ \cup \{e,e'\}; e,e'; * \rangle$ which is of finite signature. The effective encoding of $\widehat{\mathcal{H}}$ according to Lemma 9.2 yields obviously an effective encoding of $\widetilde{\mathcal{H}}$.

It is a little confusing to deal with strings over $\widetilde{\mathcal{H}}$, since their elements may be strings from $I\!N^+$. Thus, in the following discussion, we call the first ones "hyperstrings" and separate their elements (which may be strings) by the concatenation symbol " $*$ " from each other.

The sets of hyperstrings

$$A_n = \{w * w \cdot n^i \,:\, w \in I\!N^+, i \in I\!N\}, \quad n \in I\!N_+ \ ,$$

where " $w \cdot n^i$ " means the concatenation " $w \cdot n \cdots n$ " in $I\!N^+$, are $\widetilde{\mathcal{H}}$–recognizable, as one easily shows. Assume, there is an m–complete set $\bar{W}$, every A_n would be m–reducible to $\bar{W}$ via some $\widetilde{\mathcal{H}}$–computable function ϱ_n.

Let a program $\mathcal{P}_n$ computing ϱ_n use the α–functions with the indices $n_0 = n, n_1, \cdots, n_l \in I\!N$ ($l \in I\!N$). On inputs of the form $x * y$ (hyperstrings of lenth 2), all elements of current strings at some step of program $\mathcal{P}_n$ are of the form e , e' , or

$$z \cdot n_{i_1} \cdot \cdots \cdot n_{i_\lambda} \ , \quad \text{where } z \in \{x,y\},\ \lambda \in I\!N;\ i_1, \cdots, i_\lambda \in \{0, \cdots, l\}.$$

(Herein the dot " $\cdot$ " denotes concatenation within $I\!N^+$!)

Suppose that $x \neq y$ and $x, y \notin \{n_0, n_1, \cdots, n_l\}^+ \cup \{e, e'\}$. Then, on inputs $x * y$, the equality tests in the course of working of $\mathcal{P}_n$ are satisfied either for all such inputs or for none of these. Thus, for all those inputs $x * y$, the program terminates after the same number of steps, say T, and the elements of the resulting hyperstrings $\varrho_n(x * y)$ have the same term representation within $\widetilde{\mathcal{H}}$ (with x and y considered as variables).

Now we deal with the program $\bar{\mathcal{P}}$, which recognizes $\bar{W}$, applied to input strings $\varrho_n(x*y)$. Let n be unequal to all the indices of the functions $\alpha_0, \cdots, \alpha_{m_k}$ used by program $\bar{\mathcal{P}}$. The elements of current strings in the course of working according to $\bar{\mathcal{P}}$ are of the form e , e' , or

$$z \cdot n_{i_1} \cdot \cdots \cdot n_{i_\lambda} \cdot m_{j_1} \cdot \cdots \cdot m_{j_\kappa} \ , \ \text{where } \lambda \leq T \ ,$$

$z \in \{x,y\}$, $\lambda, \kappa \in I\!N$, $i_1, \cdots, i_\lambda \in \{0, \cdots, l\}$, $j_1, \cdots, j_\kappa \in \{0, \cdots, k\}$.

Let x be a number greater than $max(n_0, \cdots, n_l, m_0, m_1, \cdots, m_k)$, and take $\tau > T$. On the inputs $\varrho_n(x * x \cdot n^\tau)$, all test equations in the course of working of program $\bar{\mathcal{P}}$ are not satisfied if they are not trivial (i.e. always true). Thus, those inputs follow the same computation path as inputs $\varrho_n(x * y)$, for every number $y > x$. Since $x * x \cdot n^\tau \in A_n$, but $x * y \notin A_n$, ϱ_n cannot be an m–reduction of A_n to $\bar{W}$. □

Let the notions of m–*quasi*completeness and of an m–*quasi*reduction be straightforwardly defined. Since every string is quasiconstructible, the general halting problem with respect to a universal function is always m–quasicomplete.

Proposition 10.2 *If a structure owns a universal function, the corresponding general halting problem is m–quasicomplete. There are structures that own m–quasicomplete sets but are not effectively encodable.*

It remains to show the second assertion. We consider the structure $\widetilde{\mathcal{N}}$ from the first part of the previous proof. An m–quasicomplete set over $\widetilde{\mathcal{N}}$ is obtained by modifying the set $\widetilde{W}$ defined there in such a way that precodes of quasiprograms $\mathcal{P}$ are admitted, too. In these precodes, instead of the quasiconstants $r_1, \cdots, r_m \in I\!N$, only their indices are given at the corresponding places. To make possible a pre–universal simulation, the string $prod(\mathcal{P}, w)$, for $w = s_1 \cdots s_k$, consists of all products

$$n_1^{\alpha_1} \cdot \cdots \cdot n_l^{\alpha_l} \cdot s_\kappa \quad (1 \leq \kappa \leq k) \quad \text{and} \quad n_1^{\alpha_1} \cdot \cdots \cdot n_l^{\alpha_l} \cdot r_\mu \quad (1 \leq \mu \leq m)\,,$$

where $\alpha_\lambda \leq max(s_1, \cdots, s_k, r_1, \cdots, r_m)$. $\Box$

We remark that even on effectively encodable structures the concepts of m–completeness and m–quasicompleteness do not coincide if there is a non–constructible element s. Indeed, if W be an m–complete set of strings, then $\bar{W} = \{s \cdot w : w \in W\}$ is m–quasicomplete. $\bar{W}$ is not m–complete, since the existence of a computable m–reduction of S^+ to $\bar{W}$ would imply the constructibility of element s.

Now we are going to consider the scalar and linear structures over the reals, $\mathcal{R}_{sc}$, $\mathcal{R}_{sc,\leq}$, $\mathcal{R}_{lin}$, $\mathcal{R}_{lin,\leq}$, as well as their discrete counterparts $\mathcal{N}_{sc}$, $\mathcal{N}_{sc,\leq}$, $\mathcal{N}_{lin}$, $\mathcal{N}_{lin,\leq}$. Even if not all these structures has explicitly been specified, their definitions should be clear now.

All these structures are constructive. So, the notations with respect to quasicomputability coincide with their analogues with respect to computability.

Since $\mathcal{N}_{lin}$ and $\mathcal{N}_{lin,\leq}$ are computationally equivalent to the structure $\mathcal{N}$, they own m–complete sets (as well as universal functions). For $\mathcal{R}_{lin}$, $\mathcal{R}_{lin,\leq}$ and $\mathcal{N}_{sc,\leq}$, the problem is open.

Proposition 10.3 *The structures $\mathcal{R}_{sc}$, $\mathcal{R}_{sc,\leq}$ and $\mathcal{N}_{sc}$ don't own m–complete sets.*

We consider the case of structure $\mathcal{R}_{sc}$.

Assume, there is an m–complete set of strings, $\bar{W}$, over $\mathcal{R}_{sc}$. Let $\bar{W}$ be recognized by a program $\bar{\mathcal{P}}$ which uses the functions $\mu_{s_1}, \cdots, \mu_{s_m}$, for some $m \in I\!N$, $s_1, \cdots, s_m \in I\!R$.

For $r \in I\!R \setminus \{0, 1, -1\}$, we define the sets

$$V_r = \{x * y : \; x, y \in I\!R, \text{ and } y = r^i \cdot x, \text{ for some } i \in I\!N\}.$$

Here we again use the star "$*$" to denote the concatenation, the dot "$\cdot$" denotes the multiplication now.

It is easily seen that every set V_r is $\mathcal{R}_{sc}$–recognizable. Therefore, there is an m–reduction ϱ_r of V_r to $\bar{W}$, which is computed by some $\mathcal{R}_{sc}$–program $\mathcal{P}_r$ that uses some scalar multiplications with indices $r_0 = r, r_1, \cdots, r_l \in I\!R$. Then the current elements in the course of working of $\mathcal{P}_r$ on inputs "$x * y$" are of the form

$$r_0^{\alpha_0} \cdot \cdots \cdot r_l^{\alpha_l} \cdot z\,,$$

where $\alpha_0, \cdots, \alpha_l \in I\!N$, and $z \in \{0, 1, x, y\}$. Thus, every nontrivial equality test during the working of $\mathcal{P}_r$ holds either for finitely many "$x * y$" only, or for all those inputs which correspond to the points of some straight line in the (x, y)–plane. Therefore, disregarded a finite subset, for almost all $i \in I\!N$, the inputs "$x * r^i \cdot x$" follow the $\neq$–path within the computation tree of $\mathcal{P}_r$. This path stops after some number, say T, of steps.

Now let r be chosen in such a way that

$$r^i \neq \sum_{\mu=1}^{m} s_\mu^{\beta_\mu} \ ,$$

for all $i \in I\!N_+$ and all integers $\beta_1, \cdots, \beta_m$. This is possible by reasons of cardinality.

We consider the action of program $\bar{\mathcal{P}}$ on inputs $\varrho_r(\, x * y\,)$ obtained on the $\neq$–path of program $\mathcal{P}_r$. The current elements during this computation have always the form

$$\prod_{\lambda=0}^{l} r_\lambda^{\alpha_\lambda} \cdot \prod_{\mu=1}^{m} s_\mu^{\beta_\mu} \cdot z$$

where $\alpha_0, \cdots, \alpha_l \in \{0, 1, \cdots, T\}$, $\beta_1, \cdots, \beta_m \in I\!N$, $z \in \{0, 1, x, y\}$.

Again, every nontrivial equality test between such elements holds at most for the points of only one straight line in the (x, y)–plane. Therefore, there are exponents $i_1 \neq i_2$ with equality tests valid for the inputs "$x * r^{i_1} \cdot y$" resp. "$x * r^{i_2} \cdot y$" and having the same pairs of systems of α–exponents (considered both sides of the equations). So we obtain

$$\prod_{\lambda=0}^{l} r_\lambda^{\alpha_\lambda} \cdot \prod_{\mu=1}^{m} s_\mu^{\beta_\mu} \ = \ r^{i_1} \quad \text{and} \quad \prod_{\lambda=0}^{l} r_\lambda^{\alpha_\lambda} \cdot \prod_{\mu=1}^{m} s_\mu^{\beta'_\mu} \ = \ r^{i_2} \ ,$$

for integers α_λ, β_μ, β'_μ. This yields $r^{i_1 - i_2} = \prod_{\mu=1}^{m} s_\mu^{\beta_\mu - \beta'_\mu}$, in contrast to our supposition.

The case of structure $\mathcal{R}_{sc,\leq}$ can similarly be treated. We assume the existence of $\bar{W}$ and define V_r like above. There is a terminating path in the computation tree of program $\mathcal{P}_r$ computing an m–reduction ϱ_r of V_r to $\bar{W}$, which corresponds to continuum many segments of the straight lines defined by $y = r^i \cdot x$ in the (x, y)–plane.

Now we chose r like above and apply a program $\bar{\mathcal{P}}$ recognizing $\bar{W}$ to the output strings of such a computation path of $\mathcal{P}_r$. The vertices of the computation tree of $\bar{\mathcal{P}}$ correspond to polygonal domains in the (x, y)–plane. To recognize exactly strings from $\varrho_r(V_r)$, these domains must finally degenerate to straight–line segments. So we have representations of some r^{i_1} and r^{i_2} like above, and the related contradiction too.

For $\mathcal{N}_{sc}$, the proof is easier, since one can use prime number properties. □

We close with some remarks on the existence of Gödel functions. In a certain sense, they represent the analogue to the m–complete sets, with respect to computable functions instead of recognizable sets.

A string function $\varphi : S^+ \succ\!\!\!\rightarrow S^+$ is said to be a *Gödel function* if, to every $\mathcal{S}$–computable function ψ, there exists an $\mathcal{S}$–computable total function $\varrho : S^+ \longrightarrow S^+$ such that

$$\psi(\, pair(w, v)) \ \cong \ \varphi(\, pair(\varrho(w), v))\,.$$

By standard techniques, one shows the following results. A string function is a Gödel function iff it is universal, admits constructible programs, and satisfies the s-m-n theorem. If the structure $\mathcal{S}$ owns a universal function which admits constructible programs, then it has such one which satisfies the s-m-n theorem, and it even owns an *optimal* Gödel function, i.e., for every computable function ψ there is a translation ϱ satisfying the conditional equation above, and, moreover, $length(\varrho(w)) \leq length(w) + k$, for some constant number k.

References

[1] G. Asser, *Rekusive Wortfunktionen.* Zeitschr. f. math. Logik und Grundlagen d. Math. 6, 1960, 258–278

[2] B. Balzer, *Erkennbarkeitsbegriffe über allgemeinen Strukturen.* Diploma Thesis, E.–M.–A.–Universität Greifswald, 1996

[3] E. Bishop, *Foundations of constuctive analysis.* McGraw–Hill, New York, 1967

[4] M. Ben-Or, D. Kozen, J. Reif, *The complexity of elementary algebra and geometry.* JCSS 32, 1986, 251–264

[5] L. Blum, *Lectures on a theory of computation and complexity over the reals (or an arbitrary ring).* ICSI, Berkeley, CA, TR–89–065

[6] L. Blum, *A theory of computation and complexity over the real numbers.* ICSI, Berkeley, CA, TR–90–058

[7] L. Blum, F. Cucker, M. Shub, S. Smale, *Complexity and real computation: a manifesto.* ICSI, Berkeley, CA, TR–95–042

[8] L. Blum, M. Shub, S. Smale, *On a theory of computation and complexity over the real numbers: NP–completeness, recursive functions and universal machines.* Bulletin of the AMS 21, 1989, 1–46

[9] L. Blum, S. Smale, *The Gödel incompleteness theorem and decidability over a ring.* ICSI, Berkeley, CA, TR–90–036. also in: From Topology to Computation: Proc. of the Smalefest. Springer–Verlag, New York, Berlin, 1993, 321–339

[10] E. Börger, *Berechenbarkeit, Komplexität, Logik.* Vieweg–Verlag, Braunschweig, Wiesbaden 1992

[11] R. E. Byerly, *Ordered subrings of the reals in which output sets are recusively enumerable.* Proc. of the AMS 118, 1993

[12] F. Cucker, M. Matamala, *On digital nondeterminism.* Preprint 1993

[13] F. Cucker, M. Shub, S. Smale, *Separation of complexity classes in Koiran's weak model.* TCS 133, 1994, 3–14

[14] L. van den Dries, *Alfred Tarski's elimination theorie for real closed fields.* J. Symb. Logic 53, 1988, 7–19

[15] E. Engeler, *Algorithmic properties of structures.* Math. System Theory 1, 1967, 183–195

[16] E. Engeler, P. Läuchli, *Berechnungstheorie für Informatiker.* B. G. Teubner, Stuttgart, 1988

[17] Ju. L. Ershov, *Theorie der Numerierungen I, II, III.* Zeitschr. f. Mathem. Logik u. Grundl. d. Math., v. 19, 1973, 289–388; v. 21, 1975, 473–584; v. 23, 1977, 289–371

[18] J. E. Fenstad, *General recursion theory.* Springer–Verlag, Berlin et al., 1980

[19] M. C. Fitting, *Fundamentals of generalized recursion theory.* Studies in Logic and the Foundations of Mathematics. v. 105, North–Holland, Amsterdam, 1981

[20] H. Friedman, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory.* Logic Colloquium 1969, North–Holland, Amsterdam 1971, 361–390

[21] H. Friedman, R. Mansfield, *Algorithmic procedures.* Trans. of the AMS 332, 1992, 297–312

[22] R. O. Gandy, *Curch's thesis and principles for mechanisms.* The Kleene Symposium, ed. by J. Barwise, H. J. Keisler, K. Kunen, North–Holland PC, Amsterdam, 1980, 123–148

[23] R. O. Gandy, C. M. E. Yates (editors), *Logic Colloquium'69.* North–Holland PC, Amsterdam, London, 1971

[24] S. J. Garland, D. C. Luckham, *Program schemes, recursion schemes, and formal Languages.* JCSS 7, 1973, 119–160

[25] C. Gaßner, *On NP–completeness for linear machines.* submitted for publication

[26] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I.* Monatshefte für Math. u. Phys. 38, 1931, 173–198

[27] J. B. Goode, *Accessible telephone directories.* J. Symb. Logic 59, 1994, 92–105

[28] C. E. Gordon, *Finitistically computable functions and relations on an abstract structure (abstract).* J. Symb. Logic 36, 1971, 704

[29] A. Hemmerling, *On genuine complexity and kinds of nondeterminism.* J. Inform. Process. Cybernet. EIK 30, 1994, 77–96

[30] A. Hemmerling, *Computability and complexity over structures of finite type.* Preprint Nr. 2–1995, Preprint–Reihe Mathematik, Ernst–Moritz–Arndt–Universität, Greifswald, 1995

[31] G. T. Herman, S. D. Isard, *Computability over arbitrary fields.* J. London Math. Soc. 2, 1970, 73–79

[32] Iu. I. Janov, *The logical schemes of algorithms.* Problems of Cybernetics 1, 1960, 82–140

[33] A. J. Kfoury, R. N. Moll, M. A. Arbib, *A programming approach to computability.* Springer–Verlag, New York et al., 1982

[34] A. J. Kfoury, *Definability by programs in first–order structures.* TCS 25, 1983, 1–66

[35] A. J. Kfoury, *Definability by deterministic and non–deterministic programs (with applications to first–order dynamic logic).* Information and Control 65, 1985, 98–121

[36] K.–I. Ko, *Complexity theory of real functions.* Birkhäuser, Boston et al., 1991

[37] G. Kreisel, *Some reasons for generalising recursion theory.* in [23], 139–198

[38] D. Lacombe, *Recursion theoretic structure for relational systems.* in [23], 3–18

[39] D. C. Luckham, D. M. R. Park, M. S. Paterson, *On formalised computer programs.* JCSS 4, 1970, 220–249

[40] A. I. Malcev, *Constructive algebras I.* Russian Math. Surveys 16, 1961, 77–129

[41] A. I. Malzev, *Algorithmen und rekursive Funktionen.* Akademie–Verlag, Berlin, 1974

[42] K. Meer, *Computations over $\mathbb{Z}$ and $\mathbb{R}$: a comparision.* J. of Complexity 6, 1990, 256–263

[43] K. Meer, *A note on a $\mathbf{P} \neq \mathbf{NP}$ result for a restricted class of real machines.* J. of Complexity 8, 1992, 451–453

[44] K. Meer, *Komplexitätsbetrachtungen für reelle Maschinenmodelle.* Verlag Shaker, Aachen 1993

[45] K. Meer, C. Michaux, *A survey on real structural complexity theory.* to appear in Bulletin of the Belgian Mathematical Society

[46] N. Megiddo, *Towards a genuinely polynomial algorithm for linear programming.* SIAM J. Comp. 12, 1983, 347–353

[47] N. Megiddo, *A general NP–completeness theorem.* From Topology to Computation: Proc. of the Smalefest. Springer–Verlag, New York, Berlin, 1993, 432–442

[48] C. Michaux, *Ordered rings over which output sets are recursively enumerable.* Proc. Amer. Math. Soc. 112, 1991, 569–575

[49] C. Michaux, $\mathbf{P} \neq \mathbf{NP}$ *over the nonstandard reals implies* $\mathbf{P} \neq \mathbf{NP}$ *over* $\mathbb{R}$. TCS 133, 1994, 95–104

[50] M. L. Minsky, *Computation: finite and infinite machines.* Prentice–Hall, Inc., Englewood Cliffs, N.J., 1967

[51] Y. N. Moschovakis, *Abstract computability and invariant definability.* J. Symb. Logic 34, 1969, 605–633

[52] Y. N. Moschovakis, *Abstract first–order computability. I,II.* Trans. Amer. Math. Soc. 138, 1969, 427–504

[53] B. Poizat, *Les Petits Cailloux, Une approche modele–theorique de l'Algorithmie.* NUR AL–MANTIQ WAL–MA'RIFAH, 1995

[54] F. P. Preparata, M. I. Shamos, *Computational Geometry.* Springer–Verlag, Berlin and New York, 1985

[55] M. O. Rabin, *Computable algebra, general theory and theory of computable fields.* Trans. Amer. Math. Soc. 95, 1960, 341–360

[56] H. Rogers Jr., *Theory of recursive functions and effective computability.* McGraw–Hill, New York, 1967

[57] R. Saint John, *Output sets, halting sets and an arithmetical hierarchy for ordered substrings of the real numbers under Blum/Shub/Smale Computation.* ICSI, Berkeley, CA, TR–94–035

[58] R. Saint John, *Theory of computation for the real numbers and subrings of the real numbers following Blum/Shub/Smale.* Dissertation. University of California at Berkeley, 1995

[59] P. Schreiber, *Theorie der geometrischen Konstruktionen.* Deutscher Verlag der Wissenschaften, Berlin, 1975

[60] J. C. Shepherdson, *Computation over abstract structures: serial and parallel procedures and Friedman's effective definitional schemes.* Logic Colloquium '73. ed. by H. E. Rose and J. C. Shepherdson, North–Holland P.C., Amsterdam, 1975, 445–513

[61] J. C. Shepherdson, *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory.* Harvey Friedman's research on the foundations of mathematics. ed by L. H. Harrington et al., North Holland, Amsterdam, 1985, 285 –308

[62] J. C. Shepherdson, Mechanisms for computing over arbitrary structures. The Universal Turing Machine, A Half–Century Survey. ed. by R. Herken, Springer–Verlag, Wien, New York, 1994, 581–601

[63] J. R. Shoenfield, *Recursion theory.* Springer–Verlag, Berlin et al., 1993

[64] I. N. Soskov, *Prime computability on partial structures.* Mathem. Logic and Its Appl., ed. by D. G. Skordev, Plenum Press, New York and London, 1987, 341–350

[65] I. N. Soskov, *Definability via enumerations.* J. Symb. Logic 54, 1989, 428–440

[66] I. N. Soskov, *An external characterization of the prime computability.* Ann. Univ. Sofia 83, 1989, 89–110

[67] I. N. Soskov, *Computability by means of effectively definable schemes and definability via enumerations.* Archive for Math. Logic 29, 1990, 187–200

[68] A. A. Soskova, *An external approach to abstract data types I.* Ann. Univ. Sofia 87, 1994, no. 1

[69] A. A. Soskova, I. N. Soskov, *Effective enumerations of abstract structures.* Heyting'88, ed. by P. Petkov, Plenum Press, New York, 1990, 361–372

[70] A. Tarski, *A decision method for elementary algebra and geometry.* University of California Press, Berkeley, 1951

[71] J. Tiuryn, *A survey of the logic of effective definitions.* in: Logic of Programs, ed. by E. Engeler, Lecture Notes in CS, v. 125, 1979, 198–245

[72] K. Weihrauch, *Computability.* Springer–Verlag, Berlin et al., 1987

[73] K. Weihrauch, *A simple introduction to computable analysis.* Informatik–Berichte 171–2/1995, Fern-Univ. Hagen, 1995