On the Power of Recursive Word-Functions Without Concatenation

Jérôme Durand-Lose(✉)

Univ. Orléans, INSA Centre Val de Loire, LIFO, EA 4022, 45067 Orléans, France
jerome.durand-lose@univ-orleans.fr

Abstract. Primitive recursion can be defined on words instead of natural numbers. Up to usual encoding, primitive recursive functions coincide. Working with words allows to address words directly and not through some integer encoding (of exponential size). Considering alphabets with at least two symbols allows to relate simply and naturally to complexity theory. Indeed, the polynomial-time complexity class (as well as **NP** and exponential time) corresponds to delayed and dynamical evaluation with a polynomial bound on the size of the trace of the computation as a direct acyclic graph.

Primitive recursion in the absence of concatenation (or successor for numbers) is investigated. Since only suffixes of an input can be output, computation is very limited; e.g. pairing and unary encoding are impossible. Yet non-trivial relations and languages can be decided. Some algebraic ($\mathtt{a}^n\mathtt{b}^n$, palindromes) and non-algebraic ($\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$) languages are decidable. It is also possible to check arithmetical constrains like $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^{P(n,m)}$ with P polynomial with positive coefficients in two (or more) variables. Every regular language is decidable if recursion can be defined on multiple functions at once.

Keywords: Primitive recursion · Recursion on words · String recursion · Word-Functions

1 Introduction

Primitive recursion and general recursion (or μ-recursion) are well-known and addressed in every textbook on computability. They are based on Peano's axiomatisation of natural numbers and form a neat definition of *computable* functions over numbers. They have been studied for a century and are the topic of innumerable articles. Nowadays, computability is not anymore considered to be just about numbers but to be about any kind of information that can be represented and manipulated through textual/symbolic representations. In recent decades, the term *recursive* has been shifting to be replaced by *computable* [10] to reflect the preeminence of the computer age and to stress on operational models rather than conceptual definitions.

The present paper advocates an alternative definition of primitive recursion grounded on sequences of symbols, i.e. words, instead of numbers. Although the

Published by Springer Nature Switzerland AG 2022
Y.-S. Han and G. Vaszil (Eds.): DCFS 2022, LNCS 13439, pp. 30–42, 2022.
https://doi.org/10.1007/978-3-031-13257-5_3

definition is natural with more than one *successor*, it has not been much studied. Or rather it has been proved that all the *main properties* coincide, so that there is little interest in a less refined design.

We feel otherwise for at least two epistemological reasons. The first one is that many articles addressing recursion on words first provide an encoding of words as numbers then work on numbers. It certainly proves that words can be represented as numbers and worked upon this way (at the cost of complexity). Shouldn't it be the other way round? Numbers are represented by words and all our basic arithmetical algorithms (e.g. multiplication) are taught for decimal representation and implemented with binary-based representations. The second reason is meeting colleagues not keen on proving by induction, and instead, they introduce some numerical measure (e.g. depth of a formula) and then make a (numerical) recursion. When dealing with words, we should use induction to manipulate them (and numbers and recursion when we need counting)[1].

There are also more practical reasons for word recursion: connections to complexity theory with a natural measure of evaluations and to language theory. The first point is to note that the time complexity naturally corresponds to the size of trace of the evaluations when nothing is reevaluated (dynamic programming) and evaluated only if needed (delayed evaluation).

The connection with language theory is developed in the paper by considering recursion without concatenation/successor (preventing encoding between numbers and words as well as pairing). It already exhibits the ability to decide some non-algebraic languages and do some arithmetic checking. In the rest of the introduction, we present a brief state of the art on recursion on words, then, the complexity connection, some results on language decision without concatenation, and finally, the outline of the paper.

In the literature, the topic is referred to as *recursion on string*, *recursion on word*, *recursive word/string-functions* or *recursion on representation*. The last denomination often means a representation of natural numbers by words enumerated in shortlex/military order (length then alphabetically) leading to a non-trivial successor word-function. The literature is rather ancient (for computer science) with a peak in the 1960's. Most of the literature deal with hierarchies and, like almost everything in the field in those days, is number-centric. We concentrate on overviews and more recent and accessible papers (and in English).

The transcription by B. Kapron of notes on a course of S. Cook in Berkeley in 1967 [5] contains the m-adic notation of numbers (digits does not include 0) and relations on weak classes (including polynomial time functions, **FP**, from [4]). This paper does not exactly use word-functions: it has primitives $\{n \mapsto 10n+i\}_{0\leq i\leq 9}$ emulating concatenation on words together with number ordering.

In [6], the authors provide a survey on counterparts on words of classical results for primitive number recursion (Ackermann function, limited recursion, Grzegorczyk and loop hierarchies). They prove that everything coincides.

[1] We restrain from coining the *primitive induction* term to avoid any misunderstanding with close fields of research.

There exists research on infinite alphabet (not the case here) like [11]. Up to some encoding with numbers, it corresponds to computation over finite sequences of numbers encoded by numbers.

Variations on base functions and operators exist in mentioned papers (e.g. limited recursion for Grzegorczyk hierarchy) as well as others. A restriction to unitary word-functions is considered in [1,3,9]. To mention a more recent work, the nowhere defined function is added to primitive recursive word-functions in [7].

As expected, as soon as a class is powerful enough to provide functions to encode and decode from one setting to another, the hierarchies correspond with the number setting. This is a motivation to investigate restrained classes. As far as we know, recursion without concatenation was not investigated.

Comparing to primitive recursion on numbers, the successor function is replaced by left concatenation for every symbol and the recursion operator has to consider every possible first symbol. Various examples of word-functions are provided, some have no counterpart in the number setting like reverting a word or testing whether a word is a palindrome. An encoding of tuples of words on any alphabet as a word in a 2-symbol alphabet is provided; thus multiple recursion is not adding any power to primitive recursion and the number of symbols does not change the hierarchies and complexity classes when there are at least two symbols.

The numbers in Peanos's axiomatisation are identified with words of a 1-symbol alphabet, and so are the functions. Proving that primitive recursive functions on integers coincide is quite straightforward with the following encoding. Let $\Sigma = \{\mathtt{a}_1, \mathtt{a}_2, \cdots, \mathtt{a}_r\}$ be the alphabet, the *r-adic* encoding function of words into natural numbers: $\langle\varepsilon\rangle = 0$ and $\mathtt{a}_k \cdot w, \langle \mathtt{a}_k \cdot w\rangle = k + r \cdot \langle w\rangle$. This encoding is onto and corresponds to the ranking number (starting from 0) of the reverse of w in the shortlex order[2]. For example $\langle \mathtt{a}_{i+1} \cdot w\rangle = \langle \mathtt{a}_i \cdot w\rangle + 1$ and $\langle \mathtt{a}_j\mathtt{a}_{i+1} \cdot w\rangle = \langle \mathtt{a}_j\mathtt{a}_i \cdot w\rangle + r$.

The natural evaluation scheme of primitive recursion functions is not very efficient (especially for numbers in unary notations), so a different scheme is used to show the proximity with the Turing machine model. The *delayed dynamic evaluation scheme* of word-functions is when the functions are called by name (not value) and only the needed expressions are evaluated (delayed) and all the evaluation results are stored so that nothing is re-evaluated (dynamic programming). An evaluation is represented by a direct acyclic graph (DAG) whose nodes are calls to function evaluations. Each node is labelled with the call: expressions of the function and of the arguments and its value (if computed). The *DAG-complexity* of an evaluation of a function is the number of nodes in the DAG. The size of the input is defined by the sum of the lengths of the input words. Given the expression of the initial function, the out-degrees of the nodes are bounded by present arities; the number of edges is thus linearly bounded in the number of nodes. Nodes are atomic operations, the length of any value is

[2] The usual definition is on the reversed word, but we define it in coherence with the restriction to left concatenation.

bounded by the input size and the DAG depth. The whole description of the labelled DAG is bounded by a polynomial in the size of its complexity.

The class of polynomial-time functions (from classical complexity theory) corresponds to the class of word-functions such that the DAG-complexity of any evaluation is bounded by a polynomial in the input size. One way, given the expression of the function, it is possible to generate an algorithm that, for any input, builds the DAG and outputs the result in polynomial time. The other way, consider a Turing machine implantation of any polynomial-time algorithm together with a polynomial that bounds its execution time. It is possible to evaluate the entry size and then the polynomial, to get the result in unary and then to do a recursion on the TM simulation. The DAG-complexity is linear in the polynomial value that bounds the iteration time of the Turing machine. Although we are using unary representation, it is still polynomial in the size of the input.

This proof can be adapted to **NP** (with polynomial-size certificates) and to exponential time. Please note that there also exists syntactic characterisation of **FP** in the number setting [2].

The paper focuses on primitive recursion without concatenation. Recursion can be used to chop off initial symbols and only suffix of the input can be output preventing the existence of any pairing or encoding function. As functions, they look rather bland; but, as language deciders (as pre-images of the empty word) they prove quite rich. Some algebraic ($\mathtt{a}^n\mathtt{b}^n$, palindromes) and non-algebraic ($\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$) languages are decidable. It is also possible to check arithmetical constrains like $\mathtt{a}^n\mathtt{b}^m\mathtt{c}^{P(n,m)}$ with P polynomial with positive coefficients in two (or more) variables. As a side results, this provides non-trivial examples of unary languages.

Multiple recursion allows to define various functions in one recursion. Usually, this operator is synthesised from single recursion using some pairing function, but no such function is available without concatenation. If multiple recursion is available, any regular language can be decided. Basically, each function corresponds to a state of a finite deterministic automaton.

A rough companion python3 library was developed to manipulate primitive recursive word-functions and check our constructions. It is available at https://www.univ-orleans.fr/lifo/Members/Jerome.Durand-Lose/Recherche/Companion/2022_DCFS.tgz.

Section 2 collects all the definitions while Sect. 3 provides the expression of various usual functions. Section 4 investigates the concatenation-less primitive recursion functions as language deciders. Section 5 shows that adding multiple recursion to the concatenation-less primitive recursion functions allows to decide all the recursive languages. Concluding remarks and perspectives are gathered in Sect. 6.

2 Definitions

An *alphabet*, Σ, is a non-empty finite set: $\{\mathtt{a}_1, \mathtt{a}_2, \cdots, \mathtt{a}_r\}$ where r is its *size*. Unless otherwise specified, its size is least 2; The set of *words* are defined by

the free monoid Σ^*. Let $\cdot$ denote the concatenation operator and ε denote the empty word. Teletype fonts are used to denote symbols from Σ and math fonts to denote words. To ease notation, the concatenation symbol is often omitted, e.g. $\mathtt{aaa}$ stands for $\mathtt{a}\cdot\mathtt{a}\cdot\mathtt{a}$.

For any number k, a *k-ary (word-)function* is a function from $(\Sigma^*)^k$ to Σ^*.

The *projection* of the ith component of a tuple of size n ($1 \leq i \leq n$) is denoted π_n^i. The *identity function* is denoted id ($=\pi_1^1$). The *constant ε function* is denoted $\widehat{\varepsilon}$ (formally there is one of arity 1, others are generated with compositions and projections). For any symbol $\mathtt{a}$, the 1-ary *left concatenation* function associated with $\mathtt{a}$, is defined by: ${}_{\mathtt{a}}\cdot(w) = \mathtt{a}\cdot w = \mathtt{a}w$. The notation $\vec{x}$ denotes a vector of arguments. Sans serif fonts are used to denote functions (in lower case) and operators (capitalised).

Numbers correspond to a 1-symbol alphabet (0 corresponds to ε). The successor of n is denoted $S(n)$ (the only available left concatenation).

Composition Operator. Let j, k be positive numbers. Let g be a k-ary function and $(h_i)_{1\leq i\leq k}$ be j-ary functions. The j-ary function $f = \mathbf{Comp}(g, (h_i)_{1\leq i\leq k})$ is uniquely defined by:

$$f(\vec{x}) = g\left(h_1(\vec{x}), \cdots, h_k(\vec{x})\right)$$

where $\vec{x}$ represents j arguments.

(Single) Recursion Operator on Σ. Let k be a positive number. Let g be a k-ary function and, for each $\mathtt{a}$ of Σ, $h_{\mathtt{a}}$ be a $k{+}2$-ary function. The $k{+}1$-ary function $f = \mathbf{Rec}(g, (h_{\mathtt{a}})_{\mathtt{a}\in\Sigma})$ is uniquely defined by:

$$f(\varepsilon, \vec{y}) = g(\vec{y}) \quad \text{and}$$
$$\forall \mathtt{a} \in \Sigma, f(\mathtt{a}\cdot w, \vec{y}) = h_{\mathtt{a}}(w, f(w, \vec{y}), \vec{y})$$

where $\vec{y}$ represents k arguments and w is any word in Σ^*.

To increase readability, vertical displays of function vectors are often used for composition and recursion.

The set of *primitive recursive functions* is the smallest set of functions containing the empty-word function, left concatenation for every symbol, all the projections, and closed for the composition and the recursion operators.

From functions, *relations* are defined as the pre-image of the ε. A unary relation represents a subset of Σ^*, i.e., a *language*.

3 First Constructions

In the spirit of the next section, concatenations are avoided as much as possible. Expressions are provided for an alphabet of size 3 (or 2 when the expression is large). The generalisation to larger alphabets is straightforward.

A *test* is a function that returns ε if and only if the condition is satisfied. It is a membership test for languages and relations.

3.1 Word Manipulations

By composition, it is possible to get any function concatenating a fixed word on the left, e.g. $_{\mathtt{a}_1\mathtt{a}_2\mathtt{a}_3}\cdot = \mathbf{Comp}({}_{\mathtt{a}_1}\cdot, (\mathbf{Comp}({}_{\mathtt{a}_2}\cdot, ({}_{\mathtt{a}_3}\cdot))))$. By composing with constant empty-word function, it is possible to get any constant function, e.g. $\widehat{\mathtt{a}_1\mathtt{a}_2\mathtt{a}_3} = \mathbf{Comp}({}_{\mathtt{a}_1}\cdot, (\mathbf{Comp}({}_{\mathtt{a}_2}\cdot, (\mathbf{Comp}({}_{\mathtt{a}_3}\cdot, (\widehat{\varepsilon}))))))$.

Basic operations on words are straightforward. The 2-ary concatenation operator can be generated from composition and recursion:

$$\cdot = \mathbf{Rec}\left(\mathsf{id}, (\mathbf{Comp}\left({}_{\mathtt{a}_1}\cdot, (\pi_3^2)\right), \mathbf{Comp}\left({}_{\mathtt{a}_2}\cdot, (\pi_3^2)\right), \mathbf{Comp}\left({}_{\mathtt{a}_3}\cdot, (\pi_3^2)\right))\right).$$

Right concatenation functions can be generated as in:

$$\cdot_{\mathtt{a}_1} = \mathbf{Rec}\left(\widehat{\mathtt{a}_1}, (\mathbf{Comp}\left({}_{\mathtt{a}_1}\cdot, (\pi_2^2)\right), \mathbf{Comp}\left({}_{\mathtt{a}_2}\cdot, (\pi_2^2)\right), \mathbf{Comp}\left({}_{\mathtt{a}_3}\cdot, (\pi_2^2)\right))\right).$$

It is possible to manipulate a word as a stack/list with functions to extract the first symbol and the rest of a word:

$$\mathsf{head} = \mathbf{Rec}(\widehat{\varepsilon}, (\widehat{\mathtt{a}_1}, \widehat{\mathtt{a}_2}, \widehat{\mathtt{a}_3})) \quad \text{and} \quad \mathsf{tail} = \mathbf{Rec}(\widehat{\varepsilon}, (\pi_2^1, \pi_2^1, \pi_2^1))$$

Please note that for head, the first symbol is *consumed* by the recursion so that it has to be generated again using a concatenation. This phenomenon makes more involving if not prevent the expression of functions without concatenation. In the following, we avoid constant functions (to avoid concatenation), so that needed constants have to be provided as arguments.

The following functions act depending on the presence of $\mathtt{a}_1$ at the beginning of the first argument. The first function returns the rest of the first argument if present, the second argument otherwise. The second function returns its argument with leading $\mathtt{a}_1$ removed (if any).

$$\mathsf{suppress}^{\mathsf{else}}_{\mathtt{a}_1} = \mathbf{Rec}\left(\mathsf{id}, \left(\pi_3^1, \pi_3^3, \pi_3^3\right)\right),$$
$$\mathsf{suppress}_{\mathtt{a}_1?} = \mathbf{Comp}(\mathsf{suppress}^{\mathsf{else}}_{\mathtt{a}_1}, (\mathsf{id},\mathsf{id})).$$

The usual test for equality over numbers does not yield a test for equality but a test to decide whether one word is the reverse of the other. This is because computation in the recursion is done after the recursive call. This is invisible with numbers since in unary notation all words are palindrome.

$$\mathsf{test}_{\mathsf{reverse}} = \mathbf{Comp}\left(\mathbf{Rec}\left(\pi_1^2 \,\middle|\, \begin{array}{l} \mathbf{Comp}\left(\mathbf{Rec}\left(\mathsf{id} \,\middle|\, \begin{matrix}\pi_3^1\\ \pi_3^3\end{matrix}\right) \,\middle|\, \begin{matrix}\pi_4^2\\ \pi_4^4\end{matrix}\right) \\ \mathbf{Comp}\left(\mathbf{Rec}\left(\mathsf{id} \,\middle|\, \begin{matrix}\pi_3^3\\ \pi_3^1\end{matrix}\right) \,\middle|\, \begin{matrix}\pi_4^2\\ \pi_4^4\end{matrix}\right) \end{array}\right) \,\middle|\, \begin{matrix}\pi_2^1\\ \pi_2^2\\ \pi_2^1\end{matrix}\right).$$

This can be used to test if a word is a palindrome:

$$\mathsf{test}_{\mathsf{palindrome}} = \mathbf{Comp}\left(\mathsf{test}_{\mathsf{reverse}} \,\middle|\, \begin{matrix}\mathsf{id}\\ \mathsf{id}\end{matrix}\right).$$

It is possible to reverse a word and then test for equality:

$$\mathsf{reverse} = \mathbf{Rec}\left(\begin{array}{l|l} \widehat{\varepsilon} & \mathbf{Comp}\left(\mathbf{Rec}\left(\widehat{\mathtt{a}_1} \,\middle|\, \begin{array}{l}\mathbf{Comp}\left(\mathtt{a}_1 \cdot \middle|\pi_2^2\right)\\ \mathbf{Comp}\left(\mathtt{a}_2 \cdot \middle|\pi_2^2\right)\end{array}\right)\middle|\pi_2^2\right)\\ & \mathbf{Comp}\left(\mathbf{Rec}\left(\widehat{\mathtt{a}_2} \,\middle|\, \begin{array}{l}\mathbf{Comp}\left(\mathtt{a}_1 \cdot \middle|\pi_2^2\right)\\ \mathbf{Comp}\left(\mathtt{a}_2 \cdot \middle|\pi_2^2\right)\end{array}\right)\middle|\pi_2^2\right)\end{array}\right),$$

$$\mathsf{test}_{\mathsf{equality}} = \mathbf{Comp}\left(\mathsf{test}_{\mathsf{reverse}} \,\middle|\, \begin{array}{l}\pi_2^1\\ \mathbf{Comp}\left(\mathsf{reverse}\,\middle|\pi_2^2\right)\end{array}\right).$$

3.2 Logical Functions

Each of these if functions works like a ternary operator with a condition/test on the first argument returning the second argument if the test succeeds, otherwise the third argument. A test succeeds if it evaluates to the empty word. The most basic function just tests whether the first argument is the empty word ($\mathsf{if}_\varepsilon(\varepsilon, y, z) = y$, and $\forall x \neq \varepsilon, \mathsf{if}_\varepsilon(x, y, z) = z$). It is defined by:

$$\mathsf{if}_\varepsilon = \mathbf{Rec}\left(\pi_2^1, (\pi_4^4, \pi_4^4)\right).$$

Conjunction and disjunction operators are defined as 2-ary functions:

$$\mathsf{and}_\varepsilon = \mathbf{Comp}\left(\mathsf{if}_\varepsilon, (\pi_2^1, \pi_2^2, \pi_2^1)\right) \qquad \text{and} \qquad \mathsf{or}_\varepsilon = \mathbf{Comp}\left(\mathsf{if}_\varepsilon, (\pi_2^1, \widehat{\varepsilon}, \pi_2^2)\right).$$

If a non-ε constant is provided, the negation function can be defined by $\mathbf{Comp}\left(\mathsf{if}_\varepsilon, (\pi_2^1, \pi_2^2, \widehat{\varepsilon})\right)$. This function has arity 2 (for the constant).

The following functions use the conditions: to start with $\mathtt{a}_1$, to belong to the regular language $\mathtt{a}_1^*$, and to the language $\mathtt{a}_1^+$:

$$\begin{aligned}\mathsf{if}_{\mathtt{a}_1\Sigma^*} &= \mathbf{Rec}\left(\pi_2^2, (\pi_4^3, \pi_4^4, \pi_4^4)\right),\\ \mathsf{if}_{\mathtt{a}_1^*} &= \mathbf{Rec}\left(\pi_2^1, (\pi_4^2, \pi_4^4, \pi_4^4)\right), \text{and}\\ \mathsf{if}_{\mathtt{a}_1^+} &= \mathbf{Rec}\left(\pi_2^2, (\mathbf{Comp}\left(\mathsf{if}_{\mathtt{a}_1^*}, (\pi_4^1, \pi_4^3, \pi_4^4)\right), \pi_4^4, \pi_4^4)\right).\end{aligned}$$

3.3 Encoding and Pairing

Any word on any finite alphabet can be encoded on 2-symbol alphabet by:

$$\begin{gathered}\varepsilon \mapsto \mathtt{a}_1\mathtt{a}_1, \text{and}\\ \mathtt{a}_{i_1} \cdot \mathtt{a}_{i_2} \cdot \cdots \cdot \mathtt{a}_{i_k} \mapsto \mathtt{a}_1\mathtt{a}_2^{i_1}\mathtt{a}_1\mathtt{a}_2^{i_2}\mathtt{a}_1 \cdots \mathtt{a}_2^{i_k}\mathtt{a}_1.\end{gathered}$$

This function is primitive recursive like its decoding function as constructed below. The special value for ε has to be taken into account both in coding and decoding. The encoding is constructed by concatenating all $\mathtt{a}_1\mathtt{a}_2^i$ to a final $\mathtt{a}_1$.

$$\mathsf{encode} = \mathbf{Comp}\left(\mathsf{if}_{\varepsilon}\left|\begin{array}{l}\mathsf{id}\\ \widehat{\mathsf{a}_1\mathsf{a}_2}\\ \mathbf{Rec}\left(\widehat{\mathsf{a}_1}\left|\begin{array}{l}\mathbf{Comp}\left(_{\mathsf{a}_1\mathsf{a}_2}\cdot\left|\pi_2^2\right.\right)\\ \mathbf{Comp}\left(_{\mathsf{a}_1\mathsf{a}_2^2}\cdot\left|\pi_2^2\right.\right)\\ \mathbf{Comp}\left(_{\mathsf{a}_1\mathsf{a}_2^3}\cdot\left|\pi_2^2\right.\right)\end{array}\right.\right)\end{array}\right.\right).$$

For decoding, a new $\mathsf{a}_{|\Sigma|}$ to be rotated is concatenated on the left for each a_1 but the first. For each a_2 the function $\mathsf{rot}_{\mathsf{first}}$ rotates the first symbol of its argument.

$$\begin{array}{c}\varepsilon \mapsto \varepsilon\\ \mathsf{a}_k\cdot w \mapsto \mathsf{a}_{k \bmod \mathsf{r}+1}\cdot w\end{array} \qquad \text{and } \mathsf{rot}_{\mathsf{first}} = \mathbf{Rec}\left(\widehat{\varepsilon}\left|\begin{array}{l}\mathbf{Comp}\left(\mathsf{a}_2\cdot\left|\pi_2^1\right.\right)\\ \mathbf{Comp}\left(\mathsf{a}_3\cdot\left|\pi_2^1\right.\right)\\ \mathbf{Comp}\left(\mathsf{a}_1\cdot\left|\pi_2^1\right.\right)\end{array}\right.\right).$$

$$\mathsf{decode} = \mathbf{Comp}\left(\mathsf{if}_{\varepsilon}\left|\begin{array}{l}\mathbf{Comp}\left(\mathsf{tail}\left|\mathsf{tail}\right.\right)\\ \widehat{\varepsilon}\\ \mathbf{Comp}\left(\mathbf{Rec}\left(\widehat{\varepsilon}\left|\begin{array}{l}\mathbf{Comp}\left(_{\mathsf{a}_3}\cdot\left|\pi_2^2\right.\right)\\ \mathbf{Comp}\left(\mathsf{rot}_{\mathsf{first}}\left|\pi_2^2\right.\right)\\ \widehat{\varepsilon}\end{array}\right.\right)\middle|\mathsf{tail}\right)\end{array}\right.\right).$$

The special value for ε allows a simple pairing by concatenation.

$$\mathsf{pair} = \mathbf{Comp}\left(\cdot\left|\begin{array}{l}\mathbf{Comp}\left(\mathsf{encode}\left|\pi_2^1\right.\right)\\ \mathbf{Comp}\left(\mathsf{encode}\left|\pi_2^2\right.\right)\end{array}\right.\right).$$

To recover the first and second values of the pair, the middle $\mathsf{a}_1\mathsf{a}_1$ should be found while potential leading or ending $\mathsf{a}_1\mathsf{a}_1$ encoding ε are treated correctly. To recover the first value, the first a_1 is discarded and $\mathsf{a}_1\mathsf{a}_1$ is searched for, preserving only what is crossed.

$$\mathsf{pair}_{\mathsf{first}} = \mathbf{Comp}\left(\mathsf{decode}\left|\mathbf{Rec}\left(\widehat{\varepsilon}\left|\begin{array}{l}\mathbf{Comp}\left(\mathsf{if}_{\mathsf{a}_1\Sigma^*}\left|\begin{array}{l}\pi_2^1\\ \widehat{\mathsf{a}_1}\\ \mathbf{Comp}\left(_{\mathsf{a}_1}\cdot\left|\pi_2^2\right.\right)\end{array}\right.\right)\\ \mathbf{Comp}\left(_{\mathsf{a}_2}\cdot\left|\pi_2^2\right.\right)\\ \mathbf{Comp}\left(_{\mathsf{a}_3}\cdot\left|\pi_2^2\right.\right)\end{array}\right.\right)\right.\right).$$

To recover the second value, the first a_1 is discarded and $\mathsf{a}_1\mathsf{a}_1$ is searched for, discarding what is crossed.

$$\mathsf{pair}_{\mathsf{second}} = \mathbf{Comp}\left(\mathbf{Rec}\left(\widehat{\varepsilon}\left|\begin{array}{l}\mathbf{Comp}\left(\mathsf{if}_{\mathsf{a}_1\Sigma^*}\left|\begin{array}{l}\pi_2^1\\ \mathbf{Comp}\left(\mathsf{pair}_{\mathsf{first}}\left|\pi_2^1\right.\right)\\ \pi_2^2\end{array}\right.\right)\\ \pi_2^2\\ \pi_2^2\end{array}\right.\right)\middle|\mathsf{suppress}_{\mathsf{a}_1?}\right).$$

This paring scheme extends straightforwardly to encode any tuple.

4 Primitive Recursion Without Concatenation

Let Σ-CL-PRec be the smallest set of functions containing the empty-word function, all the projections, and closed by the composition and the primitive recursion operators on Σ^*. A direct induction shows that:

Lemma 1. *The output of any word-function in Σ-CL-PRec must be a suffix of a word in the input.*

In particular, if the input is composed only of ε, then the output is ε. This limits the computing power and even constrains language recognition: unless a non-ε constant is provided, ε is accepted. This means that if ε is not in the language, a non-empty constant has to be provided in the input.

Since logical operators do not use concatenation, the set of decidable languages/relations is closed under union, intersection and complement (with a constant).

4.1 Some Algebraic Languages Decided in Σ-CL-PRec

Palindromes. Test for palindrome p. 6 does not use concatenation. This language is algebraic, non-ambiguous but not deterministic (it cannot be recognised by deterministic push-down automata, DPDA: it has to *guess* when the middle of the w is read).

Language $\mathtt{a}_1^n\mathtt{a}_2^n$. Function $\mathsf{test}_{\mathtt{a}_1^n\mathtt{a}_2^n}$ first considers the case of input ε (accepted). Otherwise, the input is not ε and is stored as a fail value. The first symbol has to be $\mathtt{a}_1$ (otherwise fail) and then for each discarded $\mathtt{a}_1$, a function that removes one $\mathtt{a}_2$ (or fail) is used on the output.

Technical detail: $\mathsf{test}^{\mathsf{fail}}_{\mathtt{a}_1^n\mathtt{a}_2^{n+1}}$ consumes the first $\mathtt{a}_2$ to know when $\mathtt{a}_2^n$ starts; to keep balance $\mathsf{test}_{\mathtt{a}_1^n\mathtt{a}_2^n}$ consumes the first $\mathtt{a}_1$ before handling the rest of the word to $\mathsf{test}^{\mathsf{fail}}_{\mathtt{a}_1^n\mathtt{a}_2^{n+1}}$. The label fail in the name means that a fail value has to be provided as second argument. It differs from the meaning of else since the fail value might not be used to indicate failure.

$$\mathsf{test}^{\mathsf{fail}}_{\mathtt{a}_1^n\mathtt{a}_2^{n+1}} = \mathbf{Rec}\left(\left.\begin{array}{l}\mathsf{id}\\ \mathbf{Comp}\left(\mathsf{suppress}^{\mathsf{else}}_{\mathtt{a}_2}\,\middle|\,\begin{array}{l}\pi_3^2\\ \pi_3^3\end{array}\right)\\ \pi_3^1\\ \pi_3^3\end{array}\right.\right),$$

$$\mathsf{test}_{\mathtt{a}_1^n\mathtt{a}_2^n} = \mathbf{Comp}\left(\mathbf{Rec}\left(\left.\begin{array}{l}\widehat{\varepsilon}\\ \mathbf{Comp}(\mathsf{test}^{\mathsf{fail}}_{\mathtt{a}_1^n\mathtt{a}_2^{n+1}}\,|\,\begin{array}{l}\pi_3^1\\ \pi_3^3\end{array})\\ \pi_3^3\\ \pi_3^3\end{array}\right.\right)\,\middle|\,\begin{array}{l}\mathsf{id}\\ \mathsf{id}\end{array}\right).$$

If the word is not in $\mathtt{a}_1^n\mathtt{a}_2^n$, then either the fail value is used or a $\mathtt{a}_1^n\mathtt{a}_2^n$ prefix is removed leaving a non-ε word.

This language is deterministic algebraic (can be recognised by DPDA).

Language $\mathtt{a_1^n a_2^n a_1^m} \cup \mathtt{a_1^n a_2^m a_1^m}$. On a word from $\mathtt{a_1^n a_2^n a_1^m}$, $\mathsf{test}_{\mathtt{a_1^n a_2^n}}$ should return $\mathtt{a_1^m}$. So that the end of the test is carried out by removing remaining $\mathtt{a}_1$. Removing leading $\mathtt{a}_1^*$ is done with $\mathsf{suppress}_{\mathtt{a}_1^*}$. To avoid consuming one extra symbol (the first $\mathtt{a}_{\neq 1}$), one $\mathsf{suppress}_{\mathtt{a}_1?}$ is stacked for each $\mathtt{a}_1$ and then the composition is used on a copy of the input.

$$\mathsf{suppress}_{\mathtt{a}_1^*} = \mathbf{Comp}\left(\mathbf{Rec}\left(\begin{array}{|l}\widehat{\varepsilon}\\ \pi_3^3\\ \pi_3^3\end{array}\middle|\begin{array}{l}\mathbf{Comp}\left(\mathsf{suppress}_{\mathtt{a}_1?}\,\middle|\,\pi_3^2\right)\end{array}\right)\middle|\begin{array}{l}\mathsf{id}\\ \mathsf{id}\end{array}\right),$$

$$\mathsf{test}_{\mathtt{a_1^n a_2^n a_1^m}} = \mathbf{Comp}\left(\mathsf{suppress}_{\mathtt{a}_1^*}\,\middle|\,\mathsf{test}_{\mathtt{a_1^n a_2^n}}\right).$$

The language $\mathtt{a_1^n a_2^m a_1^m}$ is decided by removing all leading $\mathtt{a}_1$ and then using previous test (swapping $\mathtt{a}_1$ and $\mathtt{a}_2$): $\mathsf{test}_{\mathtt{a_1^n a_2^m a_1^m}} = \mathbf{Comp}\left(\mathsf{test}_{\mathtt{a_1^n a_2^n}}\,\middle|\,\mathsf{suppress}_{\mathtt{a}_1^*}\right)$.

Since union of decidable languages is decidable, the algebraic language $\mathtt{a_1^n a_2^m a_1^m} \cup \mathtt{a_1^n a_2^m a_1^m}$ is decidable. This language is ambiguous.

4.2 Some Non-algebraic Languages Decided in Σ-CL-PRec

Languages $\mathtt{a_1^n a_2^n a_1^n}$. Since intersection of decidable languages is decidable, the language $\mathtt{a_1^n a_2^n a_1^n} = \mathtt{a_1^n a_2^n a_1^m} \cap \mathtt{a_1^n a_2^m a_1^m}$ is decidable. This language is not algebraic. Similarly, it is possible to prove that the languages $\mathtt{a_1^n a_2^n a_1^n} \cdots \mathtt{a_1^n}$ are all decidable.

Languages $\mathtt{a_1^n a_2}^{P(n)}$ with P Polynomial with Positive Coefficients. The idea is to deal with functions that discard (or fail) the right amount of $\mathtt{a}_2$ according to the number of $\mathtt{a}_1$ for each monomial. So that the result is empty only if the sum matches.

For each monomial, a ternary function is defined. The first argument starts with $\mathtt{a_1^n a}_{\neq 1}$ to provide the value for n. The second argument is the one to remove the $\mathtt{a}_2$ from. The third argument is returned if removing is not possible.

For constant monomial 3, the function is

$$\mathsf{remove}^{\mathsf{else}}_{\mathtt{a}_2^3} = \mathbf{Comp}\left(\mathsf{suppress}^{\mathsf{else}}_{\mathtt{a}_2}\,\middle|\,\begin{array}{l}\mathbf{Comp}\left(\mathsf{suppress}^{\mathsf{else}}_{\mathtt{a}_2}\,\middle|\,\pi_2^2\right)\\ \pi_2^2\end{array}\right)$$

$$\mathsf{remove}_{\mathtt{a}_2^3} = \mathbf{Comp}\left(\mathsf{remove}^{\mathsf{else}}_{\mathtt{a}_2^3}\,\middle|\,\begin{array}{l}\pi_3^2\\ \pi_3^3\end{array}\right).$$

For the monomial $3x$, this is done x times:

$$\mathsf{remove}_{3\mathtt{a}_2^x} = \mathbf{Comp}\left(\mathbf{Rec}\left(\begin{array}{|l}\pi_3^2\\ \pi_5^4\\ \pi_5^4\end{array}\middle|\begin{array}{l}\mathbf{Comp}\left(\mathsf{remove}^{\mathsf{else}}_{\mathtt{a}_2^3}\,\middle|\,\begin{array}{l}\pi_5^2\\ \pi_5^5\end{array}\right)\end{array}\right)\middle|\begin{array}{l}\pi_3^1\\ \pi_3^1\\ \pi_3^2\\ \pi_3^3\end{array}\right).$$

For the monomial $3x^2$, it is done x times again. The function $\mathsf{remove}_{3\mathtt{a}_2^{x^2}}$ is:

$$\mathbf{Comp}\left(\begin{array}{|l}\mathbf{Rec}\left(\begin{array}{|l}\pi_3^2\\ \\ \\ \pi_5^4\\ \pi_5^4\end{array}\middle|\begin{array}{l}\mathbf{Comp}\left(\begin{array}{|l}\mathbf{Rec}\left(\begin{array}{|l}\pi_3^2\\ \pi_5^4\\ \pi_5^4\end{array}\middle|\begin{array}{l}\mathbf{Comp}\left(\mathsf{remove}_{\mathtt{a}_2^3}^{\mathsf{else}}\middle|\begin{array}{l}\pi_2^5\\ \pi_5^5\end{array}\right)\end{array}\right)\middle|\begin{array}{l}\pi_5^3\\ \pi_5^3\\ \pi_5^2\\ \pi_5^5\end{array}\right)\end{array}\right)\middle|\begin{array}{l}\pi_3^1\\ \pi_3^1\\ \pi_3^2\\ \pi_3^3\end{array}\right).$$

Even though the definition looks involving, these are just nested for loops.

It is possible to design concatenation-less functions that yield each maximal suffixes of $\mathtt{a}_1^+\mathtt{a}_2^+\mathtt{a}_3^+\cdots\mathtt{a}_m^+$ of the form $\mathtt{a}_k^+\cdots\mathtt{a}_m^+$. Hence, all the languages $\mathtt{a}_1^+\mathtt{a}_2^+\cdots\mathtt{a}_i^n\cdots\mathtt{a}_j^{P(n)}\cdots\mathtt{a}_m^+$ (for given $i\neq j$ and P) are all decidable.

Using the same tools, it is also possible to test such languages as $\mathtt{a}_1^n\mathtt{a}_2^m\mathtt{a}_2^{P(n,m)}$ with P polynomial in 2 variables with positive coefficients. More than two variables is similarly possible.

5 Regular Languages are Decidable in Σ-CL-PRec with Multiple Recursion

The multiple recursion operator is usually synthesised with the use of a pairing function, i.e. a one-to-one function from $\Sigma^*\times\Sigma^*$ to Σ^*. Yet, no such function is available without concatenation since any pairing function would have to map $\{(\mathtt{a}_1^i,\mathtt{a}_1^j)\}_{0\leq i,j<2}$ to four distinct values, but the only possible outputs are in $\{\varepsilon,\mathtt{a}_1\}$ (the suffixes). (Adding constants would no work for $\{(\mathtt{a}_1^i,\mathtt{a}_1^j)\}_{0\leq i,j<k}$ for every k.)

Lemma 2. *There is no pairing function in Σ-CL-PRec.*

Multiple Recursion Operator on Σ. Let m and k be any positive numbers. Let $(g_i)_{1\leq i\leq m}$ be k-ary functions and, for each $\mathtt{a}$ of Σ, $(h_{\mathtt{a},i})_{1\leq i\leq m}$ be $(k{+}m{+}1)$-ary functions. The $(k{+}1)$-ary functions

$$(f_i)_{1\leq i\leq m}=\mathbf{Rec}^m\Big((g_i)_{1\leq i\leq m},(h_{\mathtt{a},i})_{\mathtt{a}\in\Sigma,1\leq i\leq m}\Big)$$

are uniquely defined by $\forall i,1\leq i\leq m$:

$$\begin{aligned}f_i(\varepsilon,\vec{y})&=g_i(\vec{y})\quad\text{and}\\ \forall\mathtt{a}\in\Sigma,\quad f_i(\mathtt{a}\cdot w,\vec{y})&=h_{\mathtt{a},i}(w,f_1(w,\vec{y}),\cdots,f_m(w,\vec{y}),\vec{y})\end{aligned}$$

where $\vec{y}$ represents k arguments.

The set Σ-CL-PRec* is defined like Σ-CL-PRec, but with the addition of the closure by the recursion operators of every arity. Lemma 1 extends to Σ-CL-PRec*: the output has to be a suffix of an input.

Regular Languages are Decidable in Σ-CL-PRec*. Let L be a regular language. It is decided by some deterministic finite automaton (Q,δ,q_0,A) where

Q is finite set of state, δ is the transition table, q_0 is the initial state, and A is the set of accepting states. We suppose that $\varepsilon \in L$ (otherwise add a constant to the input and complement).

Let the 2-ary functions $(f_q)_{q\in Q}$ be defined by multiple recursion from projections by:

$$\begin{aligned}
\forall\forall q \in A, \quad & f_q(\varepsilon, w_1) = \widehat{\varepsilon}(w_1) = \varepsilon \\
\forall q \in Q\backslash A, \quad & f_q(\varepsilon, w_1) = \pi^1_1(w_1) = w_1 \\
\forall q \in Q, \quad \forall \mathtt{a} \in \Sigma, \quad & f_q(\mathtt{a}\cdot w, w_1) = \pi^i_{|Q|+2}\Big(w, (f_s(w, w_1))_{s\in Q}, w_1\Big) \\
& \qquad = f_r(w, w_1) \\
& \qquad \text{where } \delta(q, \mathtt{a}) = r \text{ and } i \text{ suitably chosen}
\end{aligned}$$

The transition table is encoded in the recursion. The following function decides L.

$$\mathsf{test}_L = \mathbf{Comp}\left(f_{q_0}, (\pi^1_1, \pi^1_1)\right)$$

6 Conclusion

Word-recursion is a rich context allowing to address words directly and to relate to complexity theory. Although forbidding concatenation seems limiting, it allows to decide non trivial languages. It is open whether all algebraic languages are decidable, and if not, which of them are not and why. More generally, a condition for a function to be (un)computable without concatenation that would rule out functions (e.g., equality) and languages is to be found.

Without concatenation it is still possible to check constrains expressed with a polynomial with positive coefficients. Although we advocate recursion on words, the range of integer languages decidable is also wide; e.g. by testing all possible splitting in two terms, the language $\{n + n^2 | n \in \mathbb{N}\}$ can be decided.

We conjecture that even though this class is restricted, there should be some undecidable properties. For example, emptiness of accepted language might be undecidable (using diophantine equations [8]).

Any function defined without concatenation, f, satisfies $|f(x_1, \cdots, x_k)| \leq \max(|x_1|, \cdots, |x_k|)$, so that this class is included in the level E_0 of the Grzegorczyk hierarchy (see [6] for definitions). Relatively to the relations/languages theses classes defined, we lack an example to show that the inclusion is strict. We conjecture that the height of recursion in the function definition provides a proper hierarchy inside the class.

Some of provided constructions rely on duplicating the input. We are wondering whether forbidding duplication leads to a non-trivial class. Otherwise, how can it be characterised?

We would like to close this article by addressing minimisation. The few operators for words in the literature are usually number representation based (related to the shortlex order) in settings where the successor is not a base function but a

non-trivial word-function. We want to avoid the influence of numbers and refuse to impose a non-trivial order on words. In the number setting, one can consider the successor function to be just a function to provide from the current one the next value to test. We propose to take that point of view: that the minimisation operator requires another word-function to generate from the current one the next word to try (starting from the empty word), without any constraint on this function (does not have to onto, one-to-one or total, as long as it is in the class). Although it seems more complex, it corresponds to the update of variables in while loops.

References

1. Asser, G.: Primitive recursive word-functions of one variable. In: Börger, E. (ed.) Computation Theory and Logic. LNCS, vol. 270, pp. 14–19. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18170-9_150
2. Bellantoni, S.J., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Comput. Complex. **2**, 97–110 (1992). https://doi.org/10.1007/BF01201998
3. Calude, C., Sântean, L.: On a theorem of günter asser. Math. Log. Q. **36**(2), 143–147 (1990)
4. Cobham, A.: The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (ed.) Studies in Logic and the Foundations of Mathematics. In: Proceedings of the 1964 International Congress, North-Holland, pp. 24–30 (1965)
5. Cook, S.A., Kapron, B.M.: A survey of classes of primitive recursive functions. Electron. Colloquium Comput. Complex. 1 (2017). https://eccc.weizmann.ac.il/report/2017/001
6. von Henke, F.W., Rose, G., Indermark, K., Weihrauch, K.: On primitive recursive wordfunctions. Computing **15**(3), 217–234 (1975). https://doi.org/10.1007/BF02242369
7. Khachatryan, M.H.: On generalized primitive recursive string functions. Math. Probl. Comput. Sci. **43**, 42–46 (2015)
8. Matiyasevich, Y.: Hilbert's tenth problem and paradigms of computation. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) CiE 2005. LNCS, vol. 3526, pp. 310–321. Springer, Heidelberg (2005). https://doi.org/10.1007/11494645_39
9. Santean, L.: A hierarchy of unary primitive recursive string-functions. In: Dassow, J., Kelemen, J. (eds.) IMYCS 1990. LNCS, vol. 464, pp. 225–233. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53414-8_45
10. Soare, R.I.: Computability and incomputability. In: Cooper, S.B., Löwe, B., Sorbi, A. (eds.) CiE 2007. LNCS, vol. 4497, pp. 705–715. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73001-9_75
11. Vučkovi, V.: Recursive word-functions over infinite alphabets. Math. Log. Q. **13**(2), 123–138 (1970)