COMPUTATION TOOLS 2026

The Seventeenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking

ISBN: 978-1-68558-379-8

April 19 - 23, 2026

Lisbon, Portugal

COMPUTATION TOOLS 2026 Editors

Emanuele Covino, Università degli Studi di Bari, Italy

COMPUTATION TOOLS 2026

Forward

The Seventeenth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (COMPUTATION TOOLS 2026), held between April 19 – 23, 2026, continued a series of events dealing with logics, algebras, advanced computation techniques, specialized programming languages, and tools for distributed computation. Mainly, the event targeted those aspects supporting context-oriented systems, adaptive systems, service computing, patterns and content-oriented features, temporal and ubiquitous aspects, and many facets of computational benchmarking.

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the COMPUTATION TOOLS 2026 technical program committee, as well as the numerous reviewers. The creation of quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to COMPUTATION TOOLS 2026. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the COMPUTATION TOOLS 2026 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope COMPUTATION TOOLS 2026 was a successful international forum for the exchange of ideas and results between academia and industry that will promote further progress in the area of computational logics, algebras, programming, tools, and benchmarking. We also hope that Lisbon provided a pleasant environment during the conference and everyone saved some time to enjoy this beautiful city.

COMPUTATION TOOLS 2026 Steering Committee

Cornel Klein, Siemens AG, Germany
Claus-Peter Rückemann, Universität Münster / DIMF / Leibniz Universität Hannover, Germany

COMPUTATIONAL TOOLS 2026 Publicity Chairs

Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
José Miguel Jiménez, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politècnica de València, Spain

COMPUTATION TOOLS 2026

Committee

COMPUTATION TOOLS 2026 Steering Committee

Cornel Klein, Siemens AG, Germany
Claus-Peter Rückemann, Universität Münster / DIMF / Leibniz Universität Hannover, Germany

COMPUTATIONAL TOOLS 2026 Publicity Chairs

Francisco Javier Díaz Blasco, Universitat Politècnica de València, Spain
Ali Ahmad, Universitat Politècnica de València, Spain
José Miguel Jiménez, Universitat Politècnica de València, Spain
Sandra Viciano Tudela, Universitat Politècnica de València, Spain

COMPUTATION TOOLS 2026 Technical Program Committee

Lorenzo Bettini, Università di Firenze, Italy
Zineddine Bettouche, TH Deggendorf (Deggendorf Institute of Technology), Germany
Ateet Bhalla, Independent Consultant, India
Narhimene Boustia, University Saad Dahlab, Blida 1, Algeria
Azahara Camacho, RTI - Real-Time Innovations, Spain
Patricia Camacho, University of Cádiz, Spain
Angelo Ciaramella, University of Naples Parthenope, Italy
Cornel Klein, Siemens AG, Germany
Emanuele Covino, Universita' di Bari, Italy
Santiago Escobar, VRAIN - Universitat Politècnica de València, Spain
Andreas Fischer, Deggendorf Institute of Technology, Germany
Shengzhong Mao, University of Manchester, UK
Roderick Melnik, Wilfrid Laurier University, Canada
Corrado Mencar, Università degli Studi di Bari Aldo Moro, Italy
Ralph Müller-Pfefferkorn, Technische Universität Dresden, Germany
Keiko Nakata, SAP SE - Potsdam, Germany
Adam Naumowicz, University of Bialystok, Poland
Cecilia E. Nugraheni, Parahyangan Catholic University, Indonesia
Alberto Policriti, University of Udine, Italy
Kristin Yvonne Rozier, Iowa State University, USA
James Tan, Singapore University of Social Sciences, Singapore
Hans Tompits, Technische Universität Wien, Austria
Prajna Upadhyay, BITS Pilani Hyderabad, India
Miroslav Velev, Aries Design Automation, USA
Tao Zheng, Orange Labs China, China

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Embedding Microcode in Sourcecode of the Programming Language O 1
Thomas Pucklitzsch, Mathias Sporer, and Anja Pucklitzsch

On the Relation between Grossone Methodology and Diagonalization of Programs 5
Emanuele Covino and Antonella Falini

Embedding Microcode in Sourcecode of the Programming Language O

Thomas Pucklitzsch, Mathias Sporer, Anja Pucklitzsch
Department of Computer Science
Duale Hochschule Sachsen
Glauchau, Germany
e-mail: thomas.pucklitzsch@dhsn.de

***Abstract*—This paper describes capability of the new programming language O, which is still under development, to embed the microcode of the underlying Octopus machine within the O-source code. The challenge is to insert a 36 bit micro-instruction into a 32 bit register and copy it into the microprogram memory to the correct location. This technology enables time critical software components to be executed up to ten times faster compared to the implementation in machine code. The paper illustrates this effect with a simple example and examines the parameters that influence the acceleration.**

Keywords-compiler; microcode; programming language; performance; Octopus

I. Introduction

Computer hardware provides gates for the execution of simple arithmetics and logic functions. The Arithmetic Logic unit (ALU) is able to add two binary numbers or compute a logic operation for every binary digit and store the result into a register. To provide more complex operations, basic hardware operations are connected, resulting in so called microprograms. The microprograms are stored in the microprogram memory and contain a sequence of data words that directly control the hardware. These sequences will be activated by the control unit of the computer. The control unit reads a machine-instruction from the memory and executes the related microprogram. The collection of every available OP-code (operation code) is called "the instruction set" of the computer and this instruction set acts as an interface between programmer and hardware. A change or extension of the microprogram results in a modification of the instruction set of the computer. However, most architectures prevent the modification of the microprogram by the programmer for security reasons by various security technologies. [1] The microcode is located on a very low level of a computer. Programming languages are used on a significant higher level. The programming language O is an unfinished language that can be modified and easily adapted to different instruction set architectures. The motivation of this paper is to present the capability of a dynamic modification of the microcode as part of the programming language O. After presenting related work, the Octopus-machine is introduced. The following sections compare the performance of using user-defined microcode versus machine code. The article concludes with an explaination of how microinstructions are embedded in O-code and how the Ossambler layer and the the Octopus-machine hardware function.

II. Related Work

In [1] a library was introduced that enables the user to change the microcode of a Intel CPU. As described in the paper, working around several security mechanisms is necessary to make the modification possible. The aim of this modification is to increase the performance of time critical operations. The hardware developer tries to prevent changes in the microprogram. To find a way around the security mechanisms and change the microprogram is very hard and a research project of its own. For example, finding the sequence of micro-commands in a x86 architecture is very hard, because micro-commands are not stored in the microprogram memory as a consectutive sequence, but the addresses are mapped with a special permutation algorithm. Without reengineering the algorithm, it is impossible to find the address of the next micro-command. In [2] a framework is introduced, which is able to analyze and patch Intel microcode. For example, an implementation of fast software breakpoints in microcode runs 1000 times faster than the solution in Assembler. This shows the potential of customized microcode.

III. The Octopus-machine

In comparison to Intel CPUs, the Octopus-machine [3] has a very simple computer architecture. It is leaning on a part of the Java virtual machine and implements a stack machine. It was first developed for educational purposes. There are implementations in Python and C, already, but very soon a soft-core for the FPGA Tung Nano 20k will be available, too. The machine was build with the purpose of easy explaining a simple HW architecture that is also able to do some useful things for students. With the Octopus software it is easy to implement your own microcode and generate an assembler for this code that we call Ossembler. Figure 1 shows a screenshot oft the Octopus software. With the help of this Ossembler it is easily possible to write code for the customized Octopus machine, and the Ossembler inserts the correct OP-codes and operates and computes all addresses. In addition to the possibility to create customized microcode, the Octopus-machine supports dynamic import of microcode. Parts of the microprogram memory are writeable through a range of addresses.

IV. Microcode vs. Machine commands

To explain the advantages of inserting customized microcode dynamically, we use a simple example. In this example we compute the Fibonacci numbers. The Fibonacci series

 ISBN: 978-1-68558-379-8

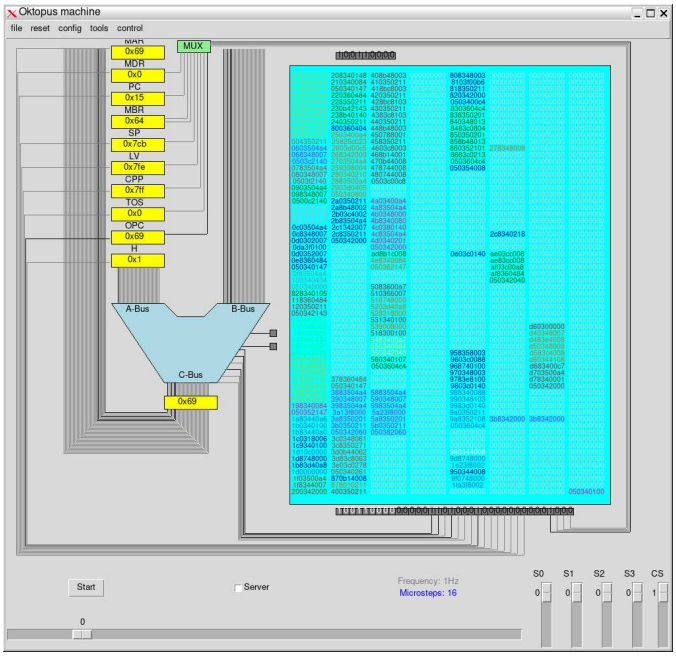

Figure 1. Octopus software

start with two ones and the next number is defined as sum of the two preceding numbers. It is not necessary to print the results because we can see the numbers appearing in the registers and the memory. To compute Fibonacci numbers, we have to add the last two numbers. A simple ossembler code is given that adds two numbers and stores the new Fibonacci number at the address of the first number.

```
oload  var2
odup
odup
:loop
oadd
odup
odup
ostore  var2
oload  var1
oadd
odup
odup
ostore  var1
oload  var2
ogoto  loop
```

Listing 1. compute Fibonacci numbers in ossembly language

Listing 1 starts with the command `oload` that reads a variable and puts it on the stack. The command `odup` doubles the top of the stack, `oadd` adds the two to elements of the stack and `ostore` saves the value on top of the stack in the given variables. The `ogoto` command jumps back to the label `:loop`. Assembly language uses so-called mnemonics insted of the operations codes which are easier for humans to handle than hexadecimal numbers. The ossembler transforms every

TABLE I. NUMBER OF CYCLES PER MICROCOMMAND

command	cycles	occurerence in loop
odup	2	4
oload	6	2
oadd	3	2
ostore	7	2
ogoto	6	1

mnemonic in an operation code that contains the address of a microprogram inside the microprogram memory that is called from the main routine. The Octopus-machine implements a typical complex instruction set architecture. That means every hardware instruction contains a different number of micro-commands. Every microcommand needs a cycle. The measure of computational cost is the number of cycles for one loop iteration. In table 1 you can see the cycles needed for every necessary kind of command and the number of occurrences in the loop cycle. With this information we can compute the number of cycles for one loop iteration. Notice that the main routine needs one cycle to call the next microprogram. With the values from table 1 and the listing we can find out that the algorithm needs 46 cycles for one loop iteration. Now we can compare the coputational costs with a microprogram that is computing the same function. In comparison with the ossembly language, a micro-program for the Octopus-machine looks like this:

1. H=MDR=1, wr
2. MAR=MDR=PC=1
3. MDR=MDR+H, wr
4. PC=MAR=PC+1
5. OPC=MDR+H
6. H=MDR
7. MDR=OPC, wr, goto 3

Listing 2. compute Fibbonacci numbers with Ossambler

The micro-commands are given in hex-code. Therefore, we show the code in a specific notation to make it better readable. The strings are registers, operations are functions of the Arithemetic Logic Unit and wr is the signal for the memory to write the content of the register MDR to the address given in the register MAR. Because every micro command contains the address of the next micro command, an additional cycle for the loop is not needed. As the Reader can see, the loop needs exactly 5 cycles for one iteration even though the algorithms are more sophisticated than the Ossembler version. As you can see the microprogram version runs nearly ten times faster than the version with machinecode. The main reason for the speedup is that results do not need to be saved to the memory between the operations. The registers hold the results until the next operation. This is the main potential of optimization.

V. MICROPROGRAM MEMORY ORGANIZATION

For inserting the microcode, the address ranges from 192 to 255 and from 448 to 511 are reserved. Figure 2 shows the microprogram memory with the two empty columns for inserting user-defined micro instuctions. The microcode allows

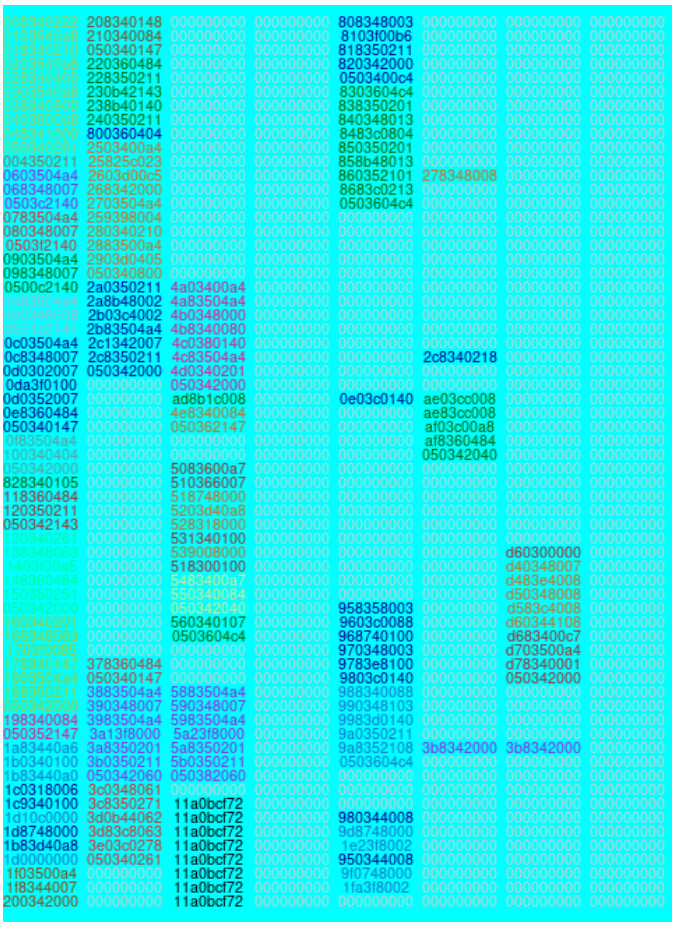

Figure 2. microprogram memory

a conditional branch. To do this, the most significant bit of the address will be inverted. This means the address of the next micro command is nextadress +/- 256. This is the reason for the two address ranges that allow a conditional branch. To write a micro command the addresses can be accessed through 0 to 127. If you write into address 0, the micro command appears in address 192 and if you write in address 64 the micro command appears in address 448. To write into the micro command, memory addresses starting from 2^{15} are being used. The problem is that micro commands for the Octopus machine need a 36 bit bus, but the memory supports 32 bit. 36 bit are needed because every micro command contains the address of the next command. To insert a micro command, two addresses must be provided. First the address where the micro command is located and second the address of the following micro command. Figure 3 shows the structure of the addresses. Because of the address ranges for dynamic micro-commands, only 128 addresses are needed. This is also useful for security purposes. It is not possible to overwrite existing microcode. Only the 128 reserved addresses are accessible. The bus has only 32 bit. The two 7 bit micro addresses are encoded into the memory address. 3 To write a micro command on address 200 in the microprogram memory with the next command on

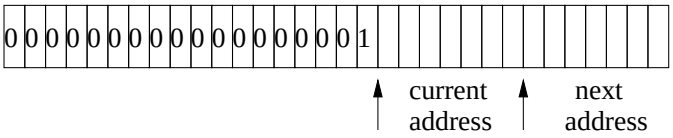

Figure 3. using memory addresses to encode two micro addresses

500, the address is computable $2^{15} + 8 + 116 * 2^7$.

VI. Integration in O and Ossambler

The predefined microcode supports a simple microprogram that takes six bytes and writes a micro command into the microprogram memory. The first four bytes contain the micro command, and the last two contain the actual address of the micro command and the address of the next micro command. The mnemonic of this microprogram is `omic`. To define a mnemonic for the new microprogram a define statement is used in Ossembler. The `define` statement needs three parameters. The mnemonic of the micro program, the address and the number of bytes the microprogram takes from the byte stream. The programming language O is a new language under construction. It is based on Extendable Makeup Language (XML) and allows a recursive structure and self explaining tags. O supports extensions. Depending on the running microcode, new tags that are keywords of O can be defined. The definition of tags can be done in the O language definition. In this definition, every tag is defined and a sequence of Ossambler commands is given. To insert microcode dynamically, a tag called `add_ microcommand` is needed. In this tag every flag of the micro command is defined with its own tag. The Reader can see the definition of a micro command in the following example.

```
<add_microcommand>
<name>new_microcommand</name>
<b-bus>MDR</b-bus>
<c-bus>TOS<c-bus>
<inc>0</inc>
<inva>1</inva>
<enb>1</enb>
<ena>1</ena>
<op>add</op>
<sra1>0</sra1>
<slla8>0</slla8>
<inc>0</inc>
<jmpn>0</jumpz>
<jmpz>1</jumpn>
<jmpc>0</jmpc>
<addr>62</addr>
<naddr></naddr>
<side>right</side>
<nside>left</nside>
</add_microcommand>
```

Listing 3. example O definition

The meaning of the different tags refers to the identical flags of the Octopus machine, which refers to the stack machine described in [4], with exception of the tags <addr>, <naddr>, <side>, <nside> and <name>. These tags provide the address where the micro-command should be stored and the address of the following micro-command. The address can be choosen from the range of 0 to 63. With the tags <side> and <nside> the row of the actual micro-command and the next micro-command can be choosen. The <name> tag defines the name of the micro-command. This name is used to call the microprogram. The actual address of the current micro-command of the example is 126 and the next address is 63. With the knowledge of the architecture of the Octopus machine it is very easy to describe the behavior of the Octopus machine in the language O.

VII. Conclusion

The Octopus machine is a simple stack-machine that allows the user to initialize the machine with an individual microprogram. It is also possible to load additional micro commands at runtime and extend the instruction set of the machine. This is possible by writing a word to the addresses of a special address range in the memory. A predefined microprogram called `omic` realizes the import of dynamic microprograms. The new programming language O is a XML-based language which is extendable. With this language it is easily possible to describe micro commands which are passed to the Ossambler for inserting into a reserved area of the microprogram memory. The possibility to define custom microcode inside the source code of the programming language O is useful for performance improving. The given example shows a speedup of factor 10, but depending on the use case the benefit can be much higher as the software breakpoints in [2] show. This means that the Octopus machine can compute many problems much faster than comparable competitors without such capabilities. Now the Octopus machine is not available as hardware, but a softcore for the use with an FPGA is under construction. The presented method makes it very easy to adapt the hardware to a specific problem in order to achieve better performance without changing the circuit.

References

[1] B. Kollenda et al., “An exploratory analysis of microcode as a building block for system defenses,” 2020.

[2] P. Borrello, C. Easdon, M. Schwarzl, R. Czerny, and M. Schwarz, “Customprocessingunit: Reverse engineering and customization of intel microcode,” 2023.

[3] T. Pucklitzsch and M. Sporer, *O - a new approach for a very simple language for distributed computation*, 2025.

[4] A. S. Tanenbaum and T. Austin, *Computerarchitektur - von der digitalen logik zum parallelrechner*, 2014.

On the Relation between Grossone Methodology and Diagonalization of Programs

Emanuele Covino
Dipartimento di Informatica
Università degli Studi di Bari
Bari, Italy
e-mail: emanuele.covino@uniba.it

Antonella Falini
Dipartimento di Informatica
Università degli Studi di Bari
Bari, Italy
e-mail: antonella.falini@uniba.it

***Abstract*—We argue that there could be a connection between the Grossone methodology, introduced by Sergeyev, and the Diagonalization operator defined in one of our previous works. The two concepts are based on similar theoretical principles, and they both deal with the problem of defining infinite objects (numbers or hierarchies of functions, respectively) with a finite number of operators.**

Keywords-grossone methodology; constructive diagonalization.

I. Introduction

Starting from year 2000, one of the Authors has investigated the link between fragments of programming languages and complexity classes [1]–[3]; these works follow the principles of the Implicit Computational Complexity theory from the 1990s. In traditional complexity, classes are defined by imposing explicit resource bounds on Turing machine's computations, but no restrictions are set on the algorithms they execute. On the other hand, implicit complexity studies classes of languages that are defined without any explicit limitation on resources, but rather by imposing suitable linguistic constraints on the way the related algorithms can be written: this allows to obtain programming languages that capture well-known complexity classes, providing what is called an implicit characterization. Within this framework, we were able to define a hierarchy of classes of programs with complexity between polynomial-time $O(n^k)$ and exponential-time $O(n^{n^k})$, using *safe composition* and *safe recursion*, and introducing a new *diagonalization* operator (they will be outlined in Section II).

Within the same timeframe, Sergeyev's work [4]–[6], focused on a new methodology of computation that allows to handle finite, infinities, and infinitesimals quantities; it is of particular interest the discussion about the relation between mechanical computation (the object of the study) and the related description (the language we use to describe it), and how the latter influences the accuracy of the obtained results. We believe that there is an intrinsic analogy between Sergeyev's *grossone* (the numeral that indicates the number of elements of the set $\mathbb{N}$ of natural numbers) and the *diagonalization* (the operator we use to jump outside an infinite sequence of classes of programs). They both share some *constructive* features: grossone is handled as a number (more precisely, as a new numeral used to describe both infinities and infinitesimals), and diagonalization is, as a matter of fact, a well defined program. They both cope with the finite description of infinite objects (numbers or classes of programs).

In Section II, we recall the definitions of safe recursion, diagonalization, and of the hierarchy of programs that can be defined using these two operators. In Section III, we recall the postulates on which the grossone methodology is based. In Section IV, we provide an insight on the previous analogy, and we outline what we consider a research path on the matter.

II. Safe recursion, diagonalization and a hierarchy of programs

In previous papers, we analyzed the computational complexity of programming languages obtained by imposing some linguistic/syntactical restrictions to their definition [2][3]. We have been able to capture a slow-growing hierarchy of programs with computing time between polynomial $O(n^k)$ and exponential $O(n^{n^k})$; this was achieved by means of two well known operators, namely *safe composition* and *safe recursion*, and introducing a new *diagonalization* operator.

Definitions of significant complexity classes have been obtained by Simmons [7], Leivant [8][9], Bellantoni and Cook [10], and Arai and Eguchi [11], among many others. In these papers, the class of polynomial-time computable functions is characterized by means of different versions of *predicative recursion* [10] or *ramified recurrence* [9], starting from a set of initial functions. A predicative definition of a recursive function is based on the idea that no explicitly bounded schemes are used, and that functions have two kind of variables: those whose values are known entirely (and which can be recursed upon), and those whose values are still being computed. These two types of variables are called *safe* and *normal* in [10] (*dormant* and *normal* in [7]); roughly speaking, normal variables are used only for recursive calls, while safe variables are used only for substitution. This separation allows to discard explicitly bounded schemes (e.g., the limited recursion) to characterize classes of functions with a given complexity (e.g., Polytime).

Our version of the *safe recursion* scheme on a binary word algebra is such that $f(x, y, za) = h(f(x, y, z), y, za)$, in which x, y and z are the auxiliary variable, the parameter, and the principal variable, respectively. In our language, the program h cannot re-assign the recursive call $f(x, y, z)$ to the principal variable z. This implies that we always know the number of recursive calls of the step program in a recursive definition: z turns out to be a *dormant* (or *safe*) variable. Starting from a natural definition of constructors and destructors over lists of binary words, we define the hierarchy of classes of programs $\mathcal{T}_k$, with $k \geq 1$, where programs in $\mathcal{T}_1$ can be computed

 ISBN: 978-1-68558-379-8

within linear time, and programs in $\mathcal{T}_{k+1}$ are obtained by one application of safe recursion to programs in $\mathcal{T}_k$; we prove that programs defined in $\mathcal{T}_k$ are exactly those programs computable within time $O(n^k)$.

In order to jump outside this infinite hierarchy of poly-time programs, an operator of *constructive diagonalization* is introduced, extending the previous hierarchy to $\mathcal{T}_\lambda$, with $\omega \leq \lambda \leq \omega^\omega$ (we follow [12] for definitions of structured ordinals). Programs in $\mathcal{T}_{\alpha+1}$ are obtained by one application of safe recursion to elements in $\mathcal{T}_\alpha$. If λ is a limit ordinal, and $\lambda_1, \ldots, \lambda_k, \ldots$ is the associated fundamental sequence, programs f in $\mathcal{T}_\lambda$ are obtained by *diagonalization* on the previously defined sequence of classes $\mathcal{T}_{\lambda_1}, \ldots, \mathcal{T}_{\lambda_k}, \ldots$, if $f(s,t) = \text{ITER}^{|t|}(f_{\lambda_{|t|}})(s,t)$, with

$$\begin{cases} \text{ITER}^1(p)(s,t) & = & \text{ITER}(p)(s,t) \\ \text{ITER}^{k+1}(p)(s,t) & = & \text{ITER}(\text{ITER}^k(p))(s,t), \end{cases}$$

and $f_{\lambda_{|t|}}$ belongs to a previously defined class $\mathcal{T}_{\lambda_{|t|}}$. $\text{ITER}(p)(s,t)$ is a simplified version of our safe recursion, that iterates the function p on s for $|t|$ times.

A program defined by diagonalization still abides by the predicative principle, given that such a program is totally constructive, it has no explicit bounds, and it basically enumerates and iterates programs that already belong to lower-level classes of programs, which are totally defined. This allows us to harmonize in a single hierarchy the classes of programs with computing time bounded by polynomial time $O(n^k)$ and exponential time $O(n^{n^k})$, for each finite k. For instance, at level ω, we select (and iterate i times) programs in the classes $\mathcal{T}_i$, where i is the length of the input; thus, the first level of diagonalization captures the class of all programs whose computation is bounded by a polynomial. By extending this approach to the next levels of structured ordinals, we were able to reach the machines computing their output within exponential time n^{n^k}.

III. GROSSONE CONCEPT

In the first years of 2000's, Yaroslav D. Sergeyev initiated a new methodology of computation in order to provide new ways of computing with infinities and infinitesimals [4]–[6]. Their approach is based on three postulates:

Postulate 1. *There exists infinite and infinitesimal objects but human beings and machines are able to execute only a finite number of operations.*

Postulate 2. *We shall not tell what are the mathematical objects we deal with; we just shall construct more powerful tools that will allow us to improve our capabilities to observe and to describe properties of mathematical objects.*

Postulate 3. *The principle 'The part is less than the whole' is applied to all numbers, (finite, infinite, or infinitesimal), and to all sets and processes (finite or infinite).*

These postulates set the basis for a new way of looking at and measuring mathematical objects, and have been applied in [4], where a new numeral system has been introduced; this system gives the possibility of dealing with numerical computations with finite, infinite, and infinitesimal numbers. In order to do this, a new infinite unit of measure has been introduced as the number of elements of the set $\mathbb{N}$ of natural numbers; it is denoted by the numeral ① , and it is called *grossone*. The Infinite Unit Axiom is what formally defines ①, and it states the principles of infinity, identity, and divisibility, as follows:

1) *infinity*: for any $n \in \mathbb{N}$, it follows that $n < ①$;
2) *identity*:
   - $0 \cdot ① = ① \cdot 0 = 0$;
   - $① - ① = 0$;
   - $\frac{①}{①} = 1$
   - $①^0 = 1^① = 1$
3) *divisibility*: for any $n \in \mathbb{N}$, the numbers $\frac{①}{n}$ are the number of elements of the n^{th} part of $\mathbb{N}$.

It is important to underline that grossone is different from Cantor's ω; due to its finite nature, grossone could be compared to our constructive diagonal operator. In [5], the reader can find a number of applications of Sergeyev's method to different areas (see [13][14] for the application of grossone methodology to the theory of computation).

IV. GROSSONE'S POSTULATES AND DIAGONALIZATION: FUTURE WORK

It is interesting to notice that the Postulates recalled in Section III do not have to be conceived as axioms in a new axiomatic system, but rather they set a methodological basis. They allow to observe and describe mathematical objects and quantities at a different level of refinement, when compared to the standard ways to handle them.

The operators of safe recursion and diagonalization introduced in Section II, together with the related hierarchy, are based on the same principles. First, there exists a potentially infinite number of programs into each class, but we use only two defining tools to capture them. Then, different ways of combining the operators allow us to deal with different hierarchies of program's classes. In our previous work, we analyze a *slow-growing* hierarchy, but we are able to characterize different complexity classes by tweaking the definition and/or the combination of the operators (for instance, by changing the way the diagonalization iterates the functions). In this sense, we are constructing tools that allow us to observe more (complexity) properties of the hierarchy.

Another analogy resides in the nature of grossone and diagonalization, because they both share the same constructive features. Grossone is defined and treated as a number (it is a new numeral used to describe both infinities and infinitesimals), and can be used into effective computations. Diagonalization is actually a well defined program that enumerates and iterates programs at a lower level of complexity; it is used to define new classes of programs that could not be defined using the safe recursion only. They both cope with the problem of describing infinite objects with finite operations.

For the previous reasons, we believe that the relation between grossone methodology and diagonalization of programs is worth exploring.

REFERENCES

[1] S. Caporaso, G. Pani, and E. Covino, "A predicative approach to the classification problem", *Journal of Functional Programming*, vol. 11, no. 1, pp. 95–116, 2001.

[2] E. Covino and G. Pani, "Diagonalization and the complexity of program", in *The Ninth International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, (COMPUTATION TOOLS 2018), February 18 - 22, 2018 – Barcelona, Spain*, 2018, pp. 1–5.

[3] E. Covino and G. Pani, "Diagonalization and Elementary Complexity", in *The 14th International Conference on Foundations of Computer Science (FCS'18), July 30 - August 2, 2018, Las Vegas, USA*, 2018, pp. 3–9.

[4] Y. D. Sergeyev, "Arithmetic of Infinity", in. Edizioni Orizzonti Meridionali, 2003.

[5] Y. D. Sergeyev, "Numerical infinity and the infinity computer", 2004, Accessed: Mar. 4, 2026. [Online]. Available: http://www.theinfinitycomputer.com

[6] Y. D. Sergeyev, "A new applied approach for executing computations with infinite and infinitesimal quantities", *Informatica*, vol. 19, no. 4, pp. 567–596, 2008.

[7] H. Simmons, "The realm of primitive recursion", *Arch.Math. Logic*, vol. 27, no. 2, pp. 177–188, 1988.

[8] D. Leivant, "Stratified functional programs and computational complexity", in *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL'93), Charleston*, 1993, pp. 325–333.

[9] D. Leivant, "Predicative recurrence and computational complexity I: word recurrence and polytime, in Feasible Mathematics II, P.Clote and J.Remmel (eds)", in. Birkauser, 1994, pp. 320–343.

[10] S. Bellantoni and S. Cook, "A New Recursion-Theoretic Characterization Of The Polytime Functions", *Computational Complexity*, vol. 2, pp. 97–110, 1992.

[11] T. Arai and N. Eguchi, "A new function algebra of EXPTIME functions by safe nested recursion", *ACM Transactions on Computational Logic*, vol. 10, no. 4, pp. 1–19, 2009.

[12] M. Fairtlough and S. Weiner, "Hierarchies of provably recursive functions, in Handbook of Proof theory, B. Samuel (ed), Studies in logic and the foundations of mathematics, vol. 137", in. Elsevier, Amsterdam, 1998, Chapter 3, pp. 149–207.

[13] Y. D. Sergeyev and A. Garro, "Observability of Turing Machines: a refinement of the theory of computation", *Informatica*, vol. 21, no. 3, pp. 425–454, 2010.

[14] Y. D. Sergeyev and A. Garro, "The Grossone methodology perspective on Turing machines", in *Automata, Universality, Computation*. A. Adamatzky (ed.), Springer Series "Emergence, Complexity and Computation", 2015, vol. 12, pp. 139-169.