On the Relation between Grossone Methodology and Diagonalization of Programs

Emanuele Covino and Antonella Falini
Dipartimento di Informatica, Universitá degli Studi di Bari, Italy
emanuele.covino@uniba.it, antonella.falini@uniba.it

A short CV

Emanuele Covino is an Assistant professor at the
Dipartimento di Informatica, Universitá degli Studi di Bari, Italy.

Research: Implicit computational complexity, Ad-hoc mobile networks,
Template metaprogramming and partial evaluation.

Teaching: Foundations of computer science, Computability and complexity,
Programming languages, Web programming,
Algorithm and data structures.

Projects: Erasmus+ Computing Competences: "Innovative learning approach for non-IT students" (agreement n° 2018-1-PL01-KA203-051143);
Horizon Europe Seeds: "Freedom of speech, new technologies, and consensus formation";
"Computational complexity of Generic programming".

A short CV

Antonella Falini is an Assistant professor at the
Dipartimento di Informatica, Universitá degli Studi di Bari, Italy.

Research: Numerical approximation, IsoGeometric analysis, Big data analytics, Numerical algorithms for data science applications.

Teaching: Numerical calculus, Numerical methods for Computer Science, Fundamentals of Mathematics for data science, Numerical algorithms.

Projects: GNCS-INdAM, "Quasi-Interpolazione Spline: *a smooth joint* per modelli differenziali, integrali e geometrici", CUP E53C25002010001;
"PON Misura Innovazione, Big Data Analytics", CUP H95F21001230006;
"PON AIM 1852414 Big Data Analytics", CUP H95G18000120006.

Table of contents

1. In our paper
2. Constructive diagonalization of programs
3. Grossone: a new methodology for infinite and infinitesimal quantities
4. Similarities and future work

In our paper

We propose a potential connection between

- the Grossone methodology, introduced by Sergeyev to handle infinite and infinitesimal computations, and
- the Diagonalization operator, defined by the Authors in previous works, to provide constructive definitions of hierarchies of functions' classes.

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, resp.) with a finite number of operators.

Constructive diagonalization of programs

Implicit Computational Complexity

- **computability theory**: what can and what cannot be computed by an algorithm, without any specific constraint on the behavior of the computational model;
- **complexity theory**: classification of computable functions based on the amount of resources used

$$\text{Turing machine} \oplus \text{time/space};$$

- **implicit computational complexity**: classes captured by imposing linguistic constraints on how algorithms are written
  - languages instead of computational models
  - what kind of constraints?
  - is there a principle shared by each different constraint?

1964 - Alan Cobham
The intrinsic computational difficulty of functions

"is it harder to multiply than to add?"

- independence from computational model and algorithm
- meta-mathematical analysis: proof systems, structure of proofs, and adequacy of systems
- meta-numerical analysis: computational systems and categories of models
- computational complexity $\Leftrightarrow$ classes of functions

. . . but which classes of functions??

the first functional characterization of Polytime

the class of functions with

- zero, successor, projections, and $2^{|x||y|}$

and closed under

- composition $f(\vec{x}, y) = h(g(\vec{x}), \ldots, g(\vec{x}))$
- bounded recursion on notation

$$\begin{cases} f(\vec{x}, 0) = g(\vec{x}) \\ f(\vec{x}, yi) = h_i(\vec{x}, y, f(\vec{x})) \text{ and } f(\vec{x}, y) \leq b(\vec{x}, y) \end{cases}$$

is the class of all functions computable within polynomial time.

Fact: the bounded recursion on notation is indecidable.

1988 - Harold Simmons
The realm of primitive recursion

$$\begin{cases} f(0, \vec{x}) = g(\vec{x}) \\ f(r+1, \vec{x}) = H(r, \vec{x}; f(r, \cdot)) \end{cases}$$

- note the ";" in H: it divides the variables in **normal** and **dormant**
- H is a functional; Simmons finds the correct class of functionals in which H is defined, in order to capture Polytime
- f is defined by (unbounded) predicative recursion.

What is a predicative definition?

a brief digression: how to define sum and product

$$\begin{cases} \oplus(0, x) = x \\ \oplus(y+1, x) = \oplus(y, x) + 1 \end{cases} \qquad \begin{cases} \otimes(0, a) = 0 \\ \otimes(b+1, a) = \oplus(a, \otimes(b, a)) \end{cases}$$

- for instance, $\otimes(3, 5) = \oplus(5, \otimes(2, 5))$;
- we can compute the $\oplus(5, \cdot)$ part, using the previous definition of $\oplus$, without knowing the value of the second variable;
- $\oplus(5, \cdot) = \oplus(4, \cdot) + 1 = \oplus(3, \cdot) + 1 + 1 = \ldots$

product can be defined differently

$$\begin{cases} \oplus(0, x) = x \\ \oplus(y+1, x) = \oplus(y, x) + 1 \end{cases}$$

$$\begin{cases} \otimes(0, a) = 0 \\ \otimes(b+1, a) = \oplus(a, \otimes(b, a)) \end{cases} \qquad \begin{cases} \otimes(0, a) = 0 \\ \otimes(b+1, a) = \oplus(\otimes(b, a), a) \end{cases}$$

- for instance, $\otimes(3, 5) = \oplus(\otimes(2, 5), 5)$;
- to compute $\oplus(\otimes(2, 5), 5)$, we need the value of $\otimes(2, 5)$;
- we are using the function $\otimes$ while defining the same function.

predicative Vs impredicative definitions

- the first definition of $\otimes$ is **predicative**
- the second one is not: we define $\otimes$ using $\otimes$

We are not surprised that in Simmons the first definition of $\otimes$ is legit, the second one is not.

Note that there isn't a predicative definition of the exponent $\uparrow(x, 2) = 2^x$

$$\begin{cases} \uparrow(0, 2) = 1 \\ \uparrow(y + 1, 2) = \otimes(\uparrow(y, 2), \uparrow(y, 2)) \end{cases}$$

1992 - Bellantoni & Cook
A new recursion-theoretic charact. of the polytime functions

Can we use Simmons' principles (separation of variable between normal and dormant) to capture Polytime?

Can we provide a predicative characterization of this class, avoiding the bounded recursion?

1992 - Bellantoni & Cook
A new recursion-theoretic charact. of the polytime functions

$$f(\underbrace{x, \ldots}_{\text{normal}}; \underbrace{y, \ldots}_{\text{safe}})$$

- initial functions: 0, $s(;a)$, $p(;a)$, $if(;a,b,c)$
- safe composition: $f(\vec{x};\vec{a}) = h(\overrightarrow{r(\vec{x};)};\overrightarrow{t(\vec{x};\vec{a})})$
- safe recursion on notation:

$$\begin{cases} f(0,\vec{x};\vec{a}) = g(\vec{x};\vec{a}) \\ f(yi,\vec{x};\vec{a}) = h_i(y,\vec{x};\vec{a},f(y,\vec{x};\vec{a})) \end{cases}$$

Polytime is the closure of the initial functions under safe composition and safe recursion on notatios, without safe inputs

- it's impossible to move variables from the safe zone to the normal one (in the definition of composition, r has no safe variables)
- hence, we cannot use the recursive call $f(y, \vec{x}; \vec{a})$ as recursive variable of another function h, also defined by recursion

We can rewrite $\oplus$ and $\otimes$ using the safe recursion; this is the only way these functions can be defined within this framework

$$\begin{cases} \oplus(0; x) = x \\ \oplus(y+1; x) = s(; \oplus(y, x)) \end{cases} \qquad \begin{cases} \otimes(0, x;) = 0 \\ \otimes(y+1, x;) = \oplus(x; \otimes(y, x;)) \end{cases}$$

- We have a predicative characterization:
  initial func's + safe recursion + safe composition = Polytime
- Can we add more definitions schemes in order to extend the previous characterization beyond Polytime, preserving the predicativity?

2018 - Covino and Pani
Diagonalization and the complexity of programs

The language is built on

- lists of binary words $Y_1 © Y_2 © \ldots © Y_n$, denoted with $r, s, \ldots$;
- the *i-th component* $(s)_i$ of a list $s = Y_1 © Y_2 © \ldots © Y_n$ is Y_i;
- $|s|$ is the length of the list s, the number of symbols occurring in s

Basic intructions are

- *constructors* $\mathrm{C}_i^a(s)$, that add the digit a at the right of the last digit of $(s)_i$, with $a = 0, 1$ and $i \geq 1$;
- *destructors* $\mathrm{D}_i(s)$, that erase the rightmost digit of $(s)_i$, with $i \geq 1$.

The program f is defined by *simple schemes* if

- f is the result of the *renaming* of x as y in g.
  Notation: $f = \mathrm{RNM}_{x/y}(g)$.
- f is the result of the *renaming* of z as y in g.
  Notation: $f = \mathrm{RNM}_{z/y}(g)$.
- f is the result of a *selection* between g and h, that is

$$f(s,t,r) = \begin{cases} g(s,t,r) & \text{if the rightmost digit} \\ & \text{of } (s)_i \text{ is } b \\ h(s,t,r) & \text{otherwise,} \end{cases}$$

  with $i \geq 1$ and $b = 0, 1$.
  Notation: $f = \mathrm{SEL}_i^b(g,h)$.

- The program f is defined by *safe composition* of h and g in the variable u if it is obtained by the substitution of h to u in g, if $u = x$ or $u = y$; the variable x must be absent in h, if $u = z$.
  Notation: $f = \mathrm{SCMP}_u(h, g)$.
- A *modifier* is a safe composition of constructors and destructors.

$\mathcal{T}_0$=(*modifier*; SCMP, SEL).
$\mathcal{T}_0$ is the closure of modifiers under selection and safe composition.

Thus, programs in $\mathcal{T}_0$ modify their inputs according to the result of some test performed over a fixed number of digits.

- The program $f(x, y, z)$ is defined by *safe recursion* in the *basis* g and in the *step* h if for all s, t, r

$$\begin{cases} f(s, t, a) & = & g(s, t) \\ f(s, t, ra) & = & h(f(s, t, r), t, ra), \end{cases}$$

  with $a = 0, 1$. Notation: $f = \mathrm{SREC}(g, h)$.

- $f(x, z)$ is defined by *iteration* of $h(x)$ if for all s, r we have

$$\begin{cases} f(s, a) & = & s \\ f(s, ra) & = & h(f(s, r)). \end{cases}$$

  with $a = 0, 1$. Notation: $f = \mathrm{ITER}(h)$.

The hierarchy of classes of programs $\mathcal{T}_k$, with $k < \omega$, is

- $\mathcal{T}_0$=(*modifier*; SCMP, SEL);
- $\mathcal{T}_1$=($\mathcal{T}_0$, ITER($\mathcal{T}_0$); SCMP, SIMPLE);
  $\mathcal{T}_1$ is the closure under safe composition and simple schemes of programs in $\mathcal{T}_0$ and in ITER($\mathcal{T}_0$);
- $\mathcal{T}_{k+1}$=($\mathcal{T}_k$, SREC($\mathcal{T}_k$); SCMP, SIMPLE);
  $\mathcal{T}_{k+1}$ is the closure under safe composition and simple schemes of programs in $\mathcal{T}_k$ and in SREC($\mathcal{T}_k$), with $k \geq 1$;

Theorem: A program f belongs to $\mathcal{T}_k$ if and only if f is computable by a register machine within time $O(n^k)$, with $k \geq 1$.

- The hierarchy $\mathcal{T}_0$, $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_k$, ..., captures the register machines that compute their output within time $O(1)$, $O(n)$, $O(n^2)$, ..., $O(n^k)$, ..., resp.;
- jumping out of the hierarchy requires more than safe recursion;
- given a limit ordinal λ, we propose a new operator that *diagonalizes* at level λ over the classes $\mathcal{T}_{\lambda_i}$, that is, that selects and iterates programs in a previously defined hierarchy of classes $\mathcal{T}_{\lambda_1}, \ldots, \mathcal{T}_{\lambda_k}$, according to the length of the input.

Given a limit ordinal λ with its fundamental sequence $\lambda_0, \ldots, \lambda_k, \ldots$, and given an enumerator program q such that $q(\lambda_i) = f_{\lambda_i}$, for each i, the program $f(x, y)$ is defined by *diagonalization* at λ if

$$f(s,t) = \text{ITER}^{|t|}(q(\lambda_{|t|}))(s,t)$$

where

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

and f_{λ_i} belongs to a previously defined class $\mathcal{C}_{\lambda_i}$, for each i.
Notation: $f = \text{DIAG}(\lambda)$.

The hierarchy of classes of programs $\mathcal{T}_\lambda$, with $\lambda < \omega^\omega$, is

- $\mathcal{T}_{\alpha+1}$=($\mathcal{T}_\alpha$, $\textsc{srec}(\mathcal{T}_\alpha)$; $\textsc{scmp}$, $\textsc{simple}$);
  $\mathcal{T}_{\alpha+1}$ is the closure under safe composition and simple schemes of programs in $\mathcal{T}_\alpha$ and programs in $\textsc{srec}(\mathcal{T}_\alpha)$;
- $\mathcal{T}_\lambda$=($\textsc{diag}(\lambda)$; $\textsc{simple}$);
  $\mathcal{T}_\lambda$ is the closure under simple schemes of programs obtained by one application of diagonalization at λ, if λ is a limit ordinal.

The *slow-growing functions* $B_\alpha : \mathbb{N} \to \mathbb{N}$ is

$$\begin{cases} B_0(n) & = & 1 \\ B_{\alpha+1}(n) & = & nB_\alpha(n) \\ B_\lambda(n) & = & B_{\lambda_n}(n). \end{cases}$$

Theorem: A program f belongs to $\mathcal{T}_\alpha$ if and only if f is computable by a register machine within time $O(B_\alpha(n))$, with $\alpha < \omega^\omega$.

Grossone: a new methodology for infinite and infinitesimal quantities

2003 - Y. D. Sergeyev
Arithmetic of Infinity (see http://www.theinfinitycomputer.com)

- a new methodology of computation with infinities and infinitesimals;
- applied to various problems (numerical computations, fractals, Turing machines, Büchi automatas);
- discussion on the relation between the observer, the mechanical computation (the object of the study) and the related description (the language we use to describe it) - the triad in natural science.

three postulates are assumed

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)*.

- The Postulates don't have to be conceived as axioms in a new logical system, but rather as a methodological basis.
- They allow to observe and describe mathematical objects at a different level of refinement, when compared to the standard ways to handle them

a new unit of measure for infinite

A new numeral ①, called *grossone*, is defined as the number of elements of the set $\mathbb{N}$.

The Infinite Unit Axiom defines ①, and it states the principles of infinity, identity, and divisibility:

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 $\mathbb{N}$.

Grossone differs from Cantor's ω; due to its finite nature, grossone could be compared to our constructive diagonal operator.

Similarities and future work

The operators of safe recursion and diagonalization, together with the related hierarchies, are based on the same three principles.

- An infinite number of programs into each class, and an infinite hierarchy of classes at each limit-ordinal level, captured by only two operators (Postulate 1).
- Different ways of tweaking the definition and/or the combination of the operators provide different levels of complexity (Postulate 2).
- The diagonal operator can iterate every function in every class at a lower level of the hierarchy, as in Postulate 3.

Grossone and diagonalization 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 a well constructed 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.

- Sergeyev builds an arithmetic hierarchy based on the new grossone numeral and its power;
- Diagonalization extends a computational hierarchy of classes already defined.

Thus, Grossone could be used instead of the diagonal operator, mimicking its behaviour, defining the proper arithmetic for new numerals ①, ②, ③, ..., and using them to return infinite sequences of classes of programs at levels ω, ω^{ω}, $\omega^{\omega^{\omega}}$, ...