A Note on a Syntactical Measure of the Complexity of Programs

Emanuele Covino
Dipartimento di Informatica, Universitá degli Studi di Bari, Italy
emanuele.covino@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, 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".

Table of contents

In our paper

- a programming language operating on stacks
- a syntactical measure σ
- a natural number $\sigma(\mathrm{P})$ assigned to each program P
- σ considers how loops defined over subprograms influences the complexity of the program
  - $\sigma(\mathrm{P}) = n \Rightarrow$ function computed by P has running time in $\mathcal{E}^{n+2}$ (the $n+2$-th Grzegorczyk class)
  - $\sigma(\mathrm{P}) = 0 \Rightarrow$ function computed by P has running time in polynomial-time

Implicit computational complexity - ICC

ICC

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

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

- **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 common principle to each constraints?

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??

1953 - A. Grzegorczyk
Some classes of recursive functions

...the candidate could be the Grzegorczyk hierarchy!

- the k-th *iterate* of f is $f^0(x) = x$ and $f^{k+1}(x) = f(f^k(x))$
- the *principal functions* $E_1, E_2, E_3, \dots$ are $E_1(x) = x^2 + 2$ and $E_{n+2}(x) = E^x_{n+1}(2)$ (the x-th iterate of E_{n+1})
- f is defined by *bounded recursion* from g, h, and b if for all $\vec{x}$, y

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

- the n-th Grzegorczyk class $\mathcal{E}^n$ is the least class of functions with functions zero, successor, projections, maximum and E_{n-1} closed under composition and bounded recursion

a few interesting facts

- $E_0(x) = x + x$
- $E_1(x) = x^2$
- $E_2(x) = x^x$
- $E_3(x) = x^{x^{\cdots^x}}$ (x times)
- $E_4(x) = x^{x^{\cdots^x}}$ ($x^{x^{\cdots^x}}$ times)
- $\ldots$

- $\mathcal{E}^0 \subseteq \mathcal{E}^1 \subseteq \mathcal{E}^2 \ldots$
- $\bigcup_i \mathcal{E}^i = \mathcal{P}R$ the primitive recursive functions

$$f \in \mathcal{E}^n$$

$$\Leftrightarrow$$

there exists a TM that computes f within space in $\mathcal{E}^n$

there exists a TM that computes f within time in $\mathcal{E}^n$

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.

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 predicative (unbounded) 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 we cannot define the exponent $\uparrow(x,2) = 2^x$ in a predicative way

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

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

Can we use the tools provided by Simmons (normal and dormant variables) to capture Polytime?

Can we give a predicative characterization, avoiding the bounded recursion?

1992 - Bellantoni & Cook
A new recursion-theoretic characterization 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
- What happens if we violate the rule of not moving variables from safe to normal zone?
  initial func's+safe recursion+safe composition+k violations $= \mathcal{E}^k$
- k violations $\Rightarrow$ the k-th Grzegorczyk class !!

Critique to this approach to complexity

Even if the safe recursion can capture Polytime, it does it via the Turing model, inefficiently

For instance

- simple sorting (polynomial) cannot be described by the safe recursion
- simple functions (the minimum) are computed with a higher complexity

insert(x, nil) $=$ cons(x,nil)
insert(x,cons(y,l)) $=$ if x$\leq$y then cons(x,cons(y,l)) else cons(x,insert(x,l))

sort(nil) $=$ nil
sort(cons(x,l)) $=$ insert(x,sort(l))

This algorithm is not defined by safe recursion

The straightforward algorithm for the minimun between two numbers is:

$$\begin{aligned}&\mathsf{min}(0,y) = 0\\&\mathsf{min}(s(x),0) = 0\\&\mathsf{min}(s(x),s(y)) = s(\mathsf{min}(x,y))\end{aligned}$$

the computation time of min is O(min(x,y)).

Defining min as a primitive recursion:

$$\mathsf{min'}(x,y) = \mathsf{if}(\mathsf{sub}(x,y),y,x)$$

the computation time of min' is O(y).

But how can I know the minimum between two numbers, if I'm still defining the minimum function?

Measures of programs

1999 - Neil Jones
LOGSPACE and PTIME characterized by programming languages

"… what is the effect of the programming style we employ
(functional, imperative, ...)
on the efficiency of the programs we can possibly write?"

Kristiansen & Niggl
On the computational complexity of imperative programming languages

An imperative programming language operating on stacks

push(a,X)
pop(X)
nil(X)

sequencing - P;Q
if-then-else - if top(X)≡a then [P]
iteration (call by value) - foreach X [P])

three examples of stack programs

P_1:= foreach X[... foreach X [push (a,Y)]]

- if v is stored in X before P_1 is executed, then Y holds $a^{|v|}$ after the execution of P_1
- the depth of loop-nesting is a necessary condition for high computational complexity, but it is not sufficient

three examples of stack programs

P_2:= nil(Y); push(a,Y); nil(Z); push(a,Z);
    foreach X [nil(Z); foreach Y [push(a,Z); push(a,Z)];
        nil(Y); foreach Z [push(a,Y)]]

P_3:= nil(Y); push(a,Y); nil(Z);
    foreach X [
        foreach Y [push(a,Z); push(a,Z); push(a,Y)]]

- both P_2 and P_3 have nesting depth 2, but
- if w is stored in X, then Z holds $a^{2^{|w|}}$ after P_2 is executed
- if w is stored in X, then Z holds $a^{|w|(|w|+1)}$ after P_3 is executed.

P_3 runs in polynomial time, whereas P_2 has exponential running time.

increasing circles

What causes the exponential growth in P_2?

P_2:= nil(Y); push(a,Y); nil(Z); push(a,Z);
foreach X [nil(Z); foreach Y [push(a,Z); push(a,Z)];
nil(Y); foreach Z [push(a,Y)]]

- there is a circle inside the outermost loop in P_2
- first Y *updates* Z (via push(a,Z))
- then Z *updates* Y
- in contrast, there is no such circle in P_3 and P_3

P_1 and P_3 both have μ measure 0; P_2 has μ measure 1

Programs with two levels of nesting circles will have μ measure 2.

a syntactical measure of the complexity of imperative programs

The μ-measure of a program operating on stacks is

μ(push(a,X))=0
μ(pop(X))=0
μ(nil(X))=0

μ(P;Q)=max(μ(P);μ(Q)))
μ(if top(X)$\equiv$a then [P])=μ(P)
μ(foreach X [P])= μ(P)+1 if there is a circle
μ(foreach X [P])= μ(P) otherwise

- programs with μ measure n can be simulated by a TM with time complexity in $\mathcal{E}^{n+2}$
- TM with time complexity in $\mathcal{E}^{n+2}$ can be simulated by programs of measure n

A note on the nature of programs - a new measure

honest and dishonest programs

- **honestly feasible programs**:
  each sub-program can be computed by a polynomial TM
- **dishonestly feasible programs**:
  - they compute a polynomial function, in more than polynomial time
  - they run in polynomial time, but some sub-program (when run separately), runs in more than polynomial time

Two lines of research

- restrict the stack program language (to capture only honest programs)
- improve the measure (to capture the highest number of program, even the dishonest)

Covino
A note on a syntactical measure of the complexity of programs

Fact: is there is a nested circle, the μ measure is increased

Questions: what happens when ...

- there are nested instructions that do not change the overall space?
- there are nested subprograms that do not change the overall space?
- there are nested circles that do not change the overall space?

Answer:

- there is no growth in the complexity of the computed function
- but the μ measure does not detect it!

introducing a new measure σ

to detect the previous situation we separate the circles in

- **increasing**, that increase the dimensions of the stacks involved in the circle itself
- **not-increasing**, that leave unchanged the total dimensions of the stacks

If a circle is not increasing, the σ measure is not increased

σ-measure for a single variable (1)

Let P be a stack program and Y a stack;
imp(Y) denotes an imperative pop(Y), push(a,Y), or nil(Y);
mod($\bar{X}$) denotes a *modifier*, i.e., a sequence of imp operating on variables in $\bar{X}$;

σ_Y(P) is defined as follow:

1. $\sigma_Y(\text{mod}(\bar{X})) := sg(\sum \hat{\sigma}_Y(\text{imp}(Y)))$, for each imp(Y) in mod($\bar{X}$), where
   $\hat{\sigma}_Y(\text{push}(a,Y)) := 1$;
   $\hat{\sigma}_Y(\text{pop}(Y)) := -1$;
   $\hat{\sigma}_Y(\text{nil}(Y)) := -\infty$;
   $\hat{\sigma}_Y(\text{imp}(X)) := 0$, with $Y \neq X$;
2. $\sigma_Y(\text{if top } Z \equiv a[P]) := \sigma_Y(P)$;
3. $\sigma_Y(P_1;P_2) := \max(\sigma_Y(P_1), \sigma_Y(P_2))$;

σ-measure for a single variable (2)

4
- $\sigma_{\mathsf{Y}}(\text{foreach X [Q]}) := \sigma_{\mathsf{Y}}(\mathsf{Q}) + 1$, if there exists a circle in Q, and a subprogram Q_i s.t.
  (a) Y and Q_i are involved in the circle;
  (b) $\sigma_{\mathsf{Y}}(\mathsf{Q}) = \sigma_{\mathsf{Y}}(\mathsf{Q}_i)$;
  (c) the circle is increasing;
- $\sigma_{\mathsf{Y}}(\text{foreach X [Q]}) := \sigma_{\mathsf{Y}}(\mathsf{Q})$, otherwise

a circle is not increasing if, denoted with Q_1,Q_2,...,Q_l and with Z_1,Z_2,...,Z_l the sequences of subprograms and, respectively, of variables involved in the circle, we have that $\sigma_{\mathsf{Z}_i}(\mathsf{Q}_j) = 0$, for each $i := 1 \ldots l$ and $j := 1 \ldots l$.

If the previous condition doesn't hold, we say that the circle is increasing.

σ-measure for a single variable (3)

Note that the "otherwise" case in (4) can be split in three different cases

1. Y is not involved in any circle
2. Y and Q_i are involved in a circle in Q, but $\sigma_Y(Q_i) \leq \sigma_Y(Q)$
   (there is a blow-up in the complexity of Y in Q_i, but this growth is lower than the growth of Y in a different subprogram of Q)
3. Y is involved in some circles, but they are not increasing
   (each variable Z_i involved in each circle doesn't produce a growth in the complexity of the subprograms Q_j involved in the same circle)

In these cases, the space used by foreach X [Q] is the same used by Q
(one can iterate a not increasing program without leading an harmful growth);
σ must remain unchanged!

σ is increased only when an increasing top circle occurs and at least one of the variables involved in that circle causes a growth in the space complexity of the related subprogram.

σ-measure for all variables

σ(P) is defined as follows:

$\sigma(\mathsf{P}) := \tilde{\sigma}(\mathsf{P}) \dot{-} 1$ (the cut-off subtraction), and

1. $\tilde{\sigma}(\mathsf{mod}(\bar{\mathsf{X}})) := 0$
2. $\tilde{\sigma}(\text{if top Z} \equiv \text{a [Q]}) := \max(\sigma_{\mathsf{Y}}(\text{if top Z} \equiv \text{a [Q]}))$, for all Y in P;
3. $\tilde{\sigma}(\mathsf{P}_1;\mathsf{P}_2) := \max(\sigma_{\mathsf{Y}}(\mathsf{P}_1;\mathsf{P}_2))$, for all Y in P;
4. $\tilde{\sigma}(\text{foreach X [Q]}) := \max(\sigma_{\mathsf{Y}}(\text{foreach X [Q]}))$, for all Y in P.

σ is better than μ for all dishonest programs

- $\sigma \leq \mu$ for all dishonest programs
- this measure considers only loops in which subprograms with a size-increasing effect are iterated

- programs with μ measure n can be simulated by a TM with time complexity in $\mathcal{E}^{n+2}$
- TM with time complexity in $\mathcal{E}^{n+2}$ can be simulated by programs of measure n