Polynomial Running Times for Polynomial-Time Oracle Machines*†

Akitoshi Kawamura[1] and Florian Steinberg[2]

1 **Graduate School of Arts and Sciences, University of Tokyo, Tokyo, Japan**
kawamura@graco.c.u-tokyo.ac.jp
2 **Department of Mathematics, Technische Universität Darmstadt, Darmstadt, Germany**
steinberg@mathematik.tu-darmstadt.de

Abstract

This paper introduces a more restrictive notion of feasibility of functionals on Baire space than the established one from second-order complexity theory. Thereby making it possible to consider functions on the natural numbers as running times of oracle Turing machines and avoiding second-order polynomials, which are notoriously difficult to handle. Furthermore, all machines that witness this stronger kind of feasibility can be clocked and the different traditions of treating partial functionals from computable analysis and second-order complexity theory are equated in a precise sense. The new notion is named 'strong polynomial-time computability', and proven to be a strictly stronger requirement than polynomial-time computability. It is proven that within the framework for complexity of operators from analysis introduced by Kawamura and Cook the classes of strongly polynomial-time computable functionals and polynomial-time computable functionals coincide.

1998 ACM Subject Classification F.1.1 Models of Computation, F.1.3 Complexity Measures and Classes

Keywords and phrases second-order complexity, oracle Turing machine, computable analysis, second-order polynomial, computational complexity of partial functionals

Digital Object Identifier 10.4230/LIPIcs.FSCD.2017.23

1 Introduction

Modern applications of second-order complexity theory almost exclusively use time-restricted oracle Turing machines to define and argue about the class of polynomial-time computable functionals [14, 7, 4, 3, 5, 12, 16, etc.]. The acceptance of this model of computation goes back to a result by Kapron and Cook [6] that characterizes the class of basic feasible functionals introduced by Mehlhorn [15].

There are several reasons for the popularity of this model of computation. Firstly, it intuitively reflects what a programmer would require of an efficient program if oracle Turing machines are interpreted as programs with subroutine calls. I.e. the time taken to evaluate the subroutine is not counted towards the time consumption (the oracle query takes one time step) and if the result is complicated the machine is given more time for further operations. Secondly, it is superficially quite close to classical polynomial-time computability: There is

* A longer version of the paper is available on the arXiv [13], https://arxiv.org/abs/1704.01405.

† This research was partly supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI JP26700001 and the Core-to-Core Program (Advanced Research Networks).

a type of functions that take sizes of the inputs and return an allowed number of steps. A subclass of these functions are considered polynomial, or 'fast' running times.

On closer inspection, however, the second-order framework introduces a whole bunch of new difficulties: Running times of oracle Turing machines, and also the functions that are considered polynomial running times, are functions of type $\mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$. These so-called second-order polynomials are a lot less well-behaved than their first-order counterparts. There are no normal-form theorems, structural induction turns out to be complicated, there is no established notion of degree and so on [10, 11]. Even worse: Second-order polynomials turn out to not be time-constructible [16].

The framework introduced by Kawamura and Cook [8] addresses this problem by restricting to length-monotone string functions, thereby forcing time-constructibility of second-order polynomials. However, it has been argued that the restriction to length-monotone string functions seems to be an unnatural one in practice [16] and that it is too restrictive to reflect some situations from practice [1]. Thus, this paper investigates different solutions to the above problems.

The content of this paper

The first part of the paper introduces the notion of polynomial-time computability from second-order complexity theory. While usually only total functionals are considered, it introduces two possible generalizations of the definition to partial functionals. The traditions of how to handle partial functionals differ a lot between computable analysis and second-order complexity theory. The two corresponding notions are proven to be distinct. This can be considered to be a very strong version of the statement that second-order polynomials are not time-constructible. Finally, it is proven that in the most important example of use of an intermediate of the two conventions, namely the framework for complexity of operators in analysis as introduced by Kawamura and Cook, could have equivalently used the convention from second-order complexity theory.

The second part of the paper presents a restriction on the behavior of oracle machines such that use of running times of higher type is not necessary anymore. It proves that the corresponding class of functionals, which are named 'strongly polynomial-time computable functionals', is a subclass of the class of polynomial-time computable functionals as defined in second-order complexity theory. It provides an example of a functional which is polynomial-time computable but not strongly polynomial-time computable. The example is not a natural, but it is pointed out without a proof that there is a candidate for a more natural example. Finally the paper presents some evidence that strong polynomial-time computability is more compatible with partial functionals and proves that within the framework for complexity of operators in analysis introduced by Kawamura and Cook, it is equivalent to polynomial-time computability.

An extended version of this paper can be found on the arXiv [13].

Conventions

Fix the finite alphabet $\Sigma := \{\mathtt{0}, \mathtt{1}\}$ and let Σ^* denote the set of finite binary strings. Elements of Σ^* are denoted by $\mathbf{a}$, $\mathbf{b}$, ... The set of non-negative integers is denoted by $\mathbb{N}$ and its elements by n, m, ... We identify the Baire space with the set $\mathcal{B} := (\Sigma^*)^{\Sigma^*}$ of string functions. Elements of $\mathcal{B}$ are denoted as φ, ψ, ... We assume the reader to be familiar with the notions of computability and complexity for elements of the Baire space introduced via Turing machines.

To compute functions on the Baire space, this paper uses oracle Turing machines: An oracle Turing machine $M^?$ is a Turing machine that has an additional oracle query tape and an oracle query state. The oracle slot may be occupied by any string function $\varphi \in \mathcal{B}$. If the computation of $M^?$ with oracle φ and input $\mathbf{a}$, also referred to as the run of M^φ on $\mathbf{a}$, enters the query state, the content of the oracle query tape, say $\mathbf{a}$, is replaced with the value $\varphi(\mathbf{a})$. For measuring time consumption, overwriting $\mathbf{a}$ with $\varphi(\mathbf{a})$ is considered to be done in one time step and the reading/writing head is not moved. The outcome of the run of $M^?$ with oracle φ on input $\mathbf{a}$ is denoted by $M^\varphi(\mathbf{a})$ and the time this computation takes by $\mathrm{time}_{M^\varphi}(\mathbf{a})$. For $A \subseteq \mathcal{B}$ an functional $F\colon A \to \mathcal{B}$ reps. a functional $\tilde{F}\colon A \times \Sigma^* \to \Sigma^*$ if for all $\varphi \in A$ and $\mathbf{a} \in \Sigma^*$ it holds that $F(\varphi)(\mathbf{a}) = M^\varphi(\mathbf{a})$ resp. $\tilde{F}(\varphi, \mathbf{a}) = M^\varphi(\mathbf{a})$.

2 Second-order complexity theory and relativization

We usually consider functionals to be of objects of type $\mathcal{B} \to \mathcal{B}$. This section is an exception as for complexity considerations it is more natural to consider oracle Turing machines to compute objects of type $\mathcal{B} \times \Sigma^* \to \Sigma^*$. Both the string and the oracle are considered input. A meaningful bound on the number of steps a machine takes should be allowed to depend on the sizes of the inputs. The size of a binary string $\mathbf{a}$ is its binary length $|\mathbf{a}| \in \mathbb{N}$. But also computations on 'big' oracles φ should be granted more time. The size of an oracle is not a natural number anymore, but a function on the natural numbers:

▶ **Definition 1.** For a string function $\varphi \in \mathcal{B}$, define its **size function** $|\varphi| : \mathbb{N} \to \mathbb{N}$ by

$$|\varphi|(n) := \max\{|\varphi(\mathbf{a})| \mid |\mathbf{a}| \leq n\}.$$

Thus, running times are objects of the type $T\colon \mathbb{N}^\mathbb{N} \times \mathbb{N} \to \mathbb{N}$: Such a function T is a running time for an oracle Turing machine $M^?$ if for any oracle φ and string $\mathbf{a}$, the run of M^φ on input $\mathbf{a}$ terminates within $T(|\varphi|, |\mathbf{a}|)$ steps. Note that it is not a priori clear what running times should be considered polynomial. The class of second-order polynomials is the smallest class of functions of type $\mathbb{N}^\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that:

- All of the functions $(l, n) \mapsto p(n)$ are contained, where p is a polynomial with natural numbers as coefficients.

And which is closed under the following operations:

- Whenever P and Q are contained, then so is their point-wise sum $P + Q$.
- Whenever P and Q are contained, then so is their point-wise product $P \cdot Q$.
- Whenever P is contained then so is the function P^+ defined by

$$P^+(l, n) := l(P(l, n)).$$

▶ **Definition 2.** A functional on Baire space is called **polynomial-time computable** if it is computed by an oracle Turing machine $M^?$ whose running time is bounded by a second-order polynomial. I.e. such that

$$\forall \varphi \in \mathcal{B}, \forall \mathbf{a} \in \Sigma^*\colon \mathrm{time}_{M^\varphi}(\mathbf{a}) \leq P(|\varphi|, |\mathbf{a}|).$$

The above definition is based on a characterization by Kapron and Cook of the class of basic feasible functionals originally introduced by Mehlhorn. Note that it is not obvious from the above definition that the class of polynomial-time computable functionals is closed under composition. To prove closure we need the following two closure properties of the set of second-order polynomials:

▶ **Lemma 3.** *Whenever P and Q are second-order polynomials, then so are*

$$(l,n) \mapsto P(Q(l,\cdot),n) \quad \text{and} \quad (l,n) \mapsto P(l,Q(l,n)).$$

The proof can be done via a tedious but straightforward induction on the term structure of second-order polynomials (an alternative proof is given in Appendix A).

▶ **Proposition 4.** *Let $F\colon \mathcal{B} \to \mathcal{B}$ and $G\colon \mathcal{B} \to \mathcal{B}$ be functionals on Baire space that can be computed within time P resp. Q. Then $F \circ G$ can be computed in time*

$$(l,n) \mapsto C \cdot (P(Q(l,\cdot),n) + Q(l,P(Q(l,\cdot),n)) \cdot P(Q(l,\cdot),n))$$

for some $C \in \mathbb{N}$. In particular, the polynomial-time computable functionals are closed under composition.

Proof (Idea). The running time bound can be obtained by combining machines that compute F and G into one that computes the composition and estimating the time this machine takes. ◀

Another property of polynomial-time computable functionals that should be mentioned is that they preserve the class of polynomial-time computable functions. This can easily be checked by combining the program of a polynomial-time machine computing the function with the program of a polynomial-time oracle Turing machine computing the functional.

Second-order polynomials were introduced as functions of type $\mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$. This is natural since they are considered running times. However, it also regularly leads to difficulties: It is not clear how to decide equality of two second-order polynomials from given construction procedures. The reader may for instance try to prove that the inequality $P \neq Q$ of two second-order polynomials as functions implies that also $P^+ \neq Q^+$. While a proof for the general case is not known to the authors, the proof becomes easy in the case where $P \neq Q$ is realized by an injective function argument. Note, that while it is not an unreasonable idea to restrict the domain of the second-order polynomials, it should at least contain all (not necessarily strictly) monotone functions, as these show up as length functions of string functions. Just like for the general case, a proof of the above if the inequality is realized by a monotone function is not known to the authors.

2.1 Relativization

Second-order complexity theory usually only considers total functionals. However, the application we are most interested in is real complexity theory, which stems from computable analysis. In computable analysis, computations on continuous structures are carried out by encoding the objects by string functions and operating on these. In this process, partial functionals are used. To see how, recall some notions from computable analysis:

▶ **Definition 5.** A **representation** ξ of a space X is a partial surjective mapping $\xi : \mathcal{B} \to X$.

An element of $\xi^{-1}(x)$ is called a **ξ-name** of x or just a **name**, if the representation is clear from the context. A pair $\mathbf{X} = (X, \xi_{\mathbf{X}})$ of a set and a representations of that set is called a represented space.

Computations on represented spaces can be carried out by operating on names:

▶ **Definition 6.** Let $f : \mathbf{X} \to \mathbf{Y}$ be a function between represented spaces. A partial functional $F :\subseteq \mathcal{B} \to \mathcal{B}$ is called a **realizer** of f if it translates $\xi_{\mathbf{X}}$-names of x to $\xi_{\mathbf{Y}}$-names of $f(x)$, that is if

$$\forall \varphi \in \mathrm{dom}(\xi_{\mathbf{X}}) : \xi_{\mathbf{Y}}(F(\varphi)) = f(\xi_{\mathbf{X}}(\varphi)).$$

A function is called **computable** if there is an oracle machine $M^?$ that computes a realizer F in the sense that $\forall \varphi \in \mathrm{dom}(\xi_{\mathbf{X}}) : M^\varphi = F(\varphi)$. Here, as is tradition in computable analysis, no assumptions are made about the behavior of the realizer outside of the domain of the representation. The characterization of polynomial-time computability of a functional by Kapron and Cook can straightforwardly be relaxed to fit this tradition.

▶ **Definition 7.** Let $A \subseteq \mathcal{B}$. We say that an oracle Turing machine $M^?$ runs in **A-restricted polynomial-time** if there exists a second-order polynomial P such that for each oracle φ from A and string $\mathbf{a}$ the computation of $M^\varphi(\mathbf{a})$ takes at most $P(|\varphi|, |\mathbf{a}|)$ steps. I.e. $\forall \varphi \in A, \forall \mathbf{a} \in \Sigma^* \colon \mathrm{time}_{M^\varphi}(\mathbf{a}) \leq P(|\varphi|, |\mathbf{a}|)$. We denote the set of functionals $F \colon A \to \mathcal{B}$ such that there is a machine computing F in A-restricted polynomial time by $\mathrm{P}(A)$.

There are two main examples of this definition covertly showing up in literature:

▶ **Example 8** (relativization)**.** Oracle machines are used in classical complexity theory to talk about polynomial-time computability of a string function $\varphi \colon \Sigma^* \to \Sigma^*$ relative to some oracle $\psi : \Sigma^* \to \{\mathtt{0}, \mathtt{1}\}$ interpreted as a subset of the strings. Under the assumption that ψ only retruns `0` or `1`, one can check that the following are equivalent:

- φ is polynomial-time computable relative to ψ.
- The constant functional returning φ is $\{\psi\}$-restricted polynomial-time computable.

Thus, the definition provides a generalization of relative polynomial-time computability, which is the reason for the name of this chapter.

The second example is Kawamura and Cook's framework for complexity for operators in analysis. Recall that Kawamura and Cook introduce the following subclass of Baire space:

▶ **Definition 9** ([8])**.** A string function $\varphi \in \mathcal{B}$ is called **length-monotone** if for all strings $\mathbf{a}$ and $\mathbf{b}$ it holds that $|\mathbf{a}| \leq |\mathbf{b}|$ implies $|\varphi(\mathbf{a})| \leq |\varphi(\mathbf{b})|$. The set of all length-monotone string functions is denoted by Σ^{**}.

Polynomial-time computability of functionals from Σ^{**} to Σ^{**} is then defined as Σ^{**}-restricted polynomial-time computability. (Of course it is not referred to by this name, but the definitions are identical.)

The tradition in second-order complexity theory is to impose the running time requirement independently of the domain of the functional.

▶ **Definition 10.** For $A \subseteq \mathcal{B}$ denote the class of all functionals $F \colon A \to \mathcal{B}$ that have a polynomial-time computable extension to all of Baire space by $\mathrm{P}|_A$.

For a partial functional $F \colon A \subseteq \mathcal{B} \to \mathcal{B}$ there are now two approaches to define polynomial-time computability. On one hand, one could require that F is A-restricted polynomial-time computable, i.e., $F \in \mathrm{P}(A)$. On the other hand, one could use the more restrictive definition that F has a total polynomial-time computable extension, i.e., $F \in \mathrm{P}|_A$. The first definition fits into the tradition of computable analysis, where usually no assumptions about a realizer are made outside of the domain of the representation on the input side of the operator. The second definition is in the tradition of second-order complexity theory, where one usually only considers polynomial-time computability of total functionals.

Note, that this situation is already encountered in classical complexity theory: For a partial function $\varphi : \Omega \subseteq \Sigma^* \to \Sigma^*$, polynomial time computability can be defined by requiring the existence of a machine M that computes φ and a polynomial p such that either

$$\forall \mathbf{a} \in \Omega \colon \mathrm{time}_M(\mathbf{a}) \leq p(a) \quad \text{or} \quad \forall \mathbf{a} \in \Sigma^* \colon \mathrm{time}_M(\mathbf{a}) \leq p(a).$$

Due to the time-constructibility of polynomials, these choices lead to the same class of partial functions. This argument does not generalize to the second-order framework.

2.2 Incompatibility with relativization

Of course, the distinction between $\mathrm{P}(A)$ and $\mathrm{P}\,|_A$ only makes sense if these classes differ in general. Note that by definition $\mathrm{P}(A) \supseteq \mathrm{P}\,|_A$. Before we give the example that separates the classes, let us discuss why this result is not obvious. The basic idea is to consider the length function on the string functions. Any oracle machine that computes this function takes a minimum of 2^n steps on any input of length n and arbitrary oracle, as each query of length n has to be asked to guarantee correctness of the return value. On the other hand, the brute-force search computes the length function in about $2^n l(n)$ time steps. This means, that the length function becomes A-restricted polynomial-time computable if A is chosen as the set of string functions that have at least exponential length.

Why does this not provide a counterexample already? Unfortunately, the brute force search can be modified to detect names of subexponential length and abort the computation in time. Informally such an algorithm can be described as follows: 'Do a brute-force search, but abort as soon as you have to ask more than twice as many oracle queries as the length of the biggest return value you have found so far'. Such a machine does indeed compute the restriction of the length function on the exponentially growing functions while running in polynomial time for all inputs and returning something that differs from the length on the shorter functions (this is allowed since they are not in the domain).

Thus, the argument has to be more elaborate. Our solution is to delay the time until a big return values are guaranteed: The elements of A are only required to exhibit exponential growth on a sparse subset, i.e. $|\varphi|\,(g(n)) \geq 2^{g(n)}$, where g is a fast growing function. Note that if g does not grow fast enough, the trick above does still work. For instance for $g(n) = 2^n$, the following algorithm works: 'Do a brute-force search but abort as soon as you have to ask more queries than the square of the biggest return value you have found so far'. If g grows too fast the A-restricted polynomial-time computability may break down.

Fortunately the choice $g(n) = 2^{2^n}$ is a sweet spot: On one hand, due to the availability of length function iteration, it is still possible to extract a super exponential function from an element of the set therefore to make the brute-force algorithm work in A-restricted polynomial-time. On the other hand the above approach to compute a total extension does not work anymore and it becomes provable that no polynomial-time computable extension exists.

▶ **Theorem 11** (in general $\mathrm{P}\,|_A \subsetneq \mathrm{P}(A)$)**.** *There exist a set $A \subseteq \mathcal{B}$ and a functional $F : A \to \mathcal{B}$ such that F is A-restricted polynomial-time computable but has no total polynomial-time computable extension.*

Proof. Consider the set

$$A := \{\varphi \in \mathcal{B} \mid \forall n \in \mathbb{N} : |\varphi|\,(2^{2^n}) \geq 2^{2^{2^n}}\}$$

and the functional on A defined by

$$F : A \to \mathcal{B}, \quad F(\varphi)(\mathbf{a}) := 0^{|\varphi|(|\mathbf{a}|)}.$$

F is A-restricted polynomial-time computable. To see that this is true first note that $3(n+2) \geq 2^{\lceil \mathrm{lb}(\mathrm{lb}(3(n+2)))\rceil - 1} \in \mathbb{N}$ and $\mathrm{lb}(n)^2 \leq 3(n+2)$ (this is implied by the inequality $\ln(x) \leq \frac{x-1}{\sqrt{x}}$). Thus, for $\varphi \in A$ it holds that

$$\begin{aligned}|\varphi|\,(|\varphi|\,(3(n+2))) &\geq |\varphi|\,(|\varphi|\,(2^{2^{\lceil \mathrm{lb}(\mathrm{lb}(3(n+2)))\rceil - 1}})) \geq |\varphi|\,(2^{2^{2^{\lceil \mathrm{lb}(\mathrm{lb}(3(n+2)))\rceil - 1}}}) \\ &\geq 2^{2^{2^{2^{\lceil \mathrm{lb}(\mathrm{lb}(3(n+2)))\rceil - 1}}}} \geq 2^{2^{\sqrt{3(n+2)}}} \geq 2^n\end{aligned}$$

This means that a second-order polynomial provides sufficient time to find the value of $|\varphi|(|\mathbf{a}|)$ in A-restricted polynomial time using a brute-force search.

However, F does not have a total polynomial-time computable extension, as can be seen as follows: Towards a contradiction assume that there is an oracle Turing machine $M^?$ that computes such an extension in time bounded by some second-order polynomial P. For each $n \in \mathbb{N}$ define an oracle $\varphi_n \in A$. First define a sequence of functions $\varphi_{n,k} \in \mathcal{B}$. Let $\varphi_{n,0}$ be the constant function returning ε. To recursively define $\varphi_{n,k+1}$ follow the computation $M^{\varphi_{n,k}}(0^{2^{2^n}-1})$ and whenever a query $\mathbf{a}$ is asked such that $\mathrm{lb}(\mathrm{lb}(|\mathbf{a}|))$ is an integer, then check whether all other queries of this length have been asked before and were answered with an ε by $\varphi_{n,k}$. If this situation is encountered for some query $\mathbf{a}_k$, then set $\varphi_{n,k+1}(\mathbf{a}_k)$ to be the string of $2^{2^{2^m}}$ zeros, ignore the rest of the computation and set for $\varphi_{n,k+1}(\mathbf{b}) := \varphi_{n,k}(\mathbf{b})$ for all other strings. If such an $\mathbf{a}_k$ does not exist, then set $\varphi_{n,k+1} := \varphi_{n,k}$. The sequence $(\varphi_{n,k})_k$ converges in Baire space, as the sequence $\varphi_{n,k}(\mathbf{a})$ is either constantly ε or jumps to $0^{2^{|\mathbf{a}|}}$ at some point and remains constant afterwards. Let $\tilde{\varphi}_n$ be the limit. Since $M^?$ is a deterministic machine, the computations $M^{\tilde{\varphi}_n}(0^{2^{2^n}-1})$ and $M^{\varphi_{n,k}}(0^{2^{2^n}-1})$ are identical up until the query $\mathbf{a}_{k+1}$ is done. For the computation on oracle $\tilde{\varphi}_n$ to be finite, the sequence $(\mathbf{a}_k)_k$ must be finite. Let k_0 be bigger than the number of elements, then $\tilde{\varphi}_n = \varphi_{n,k_0}$. Let φ_n be the function that is identical to φ_{n,k_0} unless φ_{n,k_0} returns ε on all inputs of length 2^{2^m}. In this case not all the queries of this length were asked in the run of the machine $M^?$ on oracle φ_{n,k_0} and input $0^{2^{2^n}-1}$. Pick one query of length 2^{2^m} that was not asked and let φ_n return the string of $2^{2^{2^m}}$ zeros on this string. This guarantees that $\varphi_n \in A$.

Let ψ_n be the string function that coincides with φ_n on strings of length less or equal $2^{2^{n-1}}$ (and thus also on all strings of length less or equal $2^{2^n}-1$) and returns ε on bigger strings. Since the machine $M^?$ is deterministic for $M^{\varphi_n}(0^{2^{2^n}-1})$ and $M^{\psi_n}(0^{2^{2^n}-1})$ to differ it is necessary that an oracle query has been asked such that the answers of ψ_n and φ_n are distinct. The definition of φ_n makes sure that this does not happen before all queries of length 2^{2^n} have been posed. Each of these queries takes one time step, thus $\mathrm{time}_{M^{\psi_n}}(\mathbf{a}) \geq 2^{2^{2^n}}$ if the return values differ. If the machine runs identically on oracle φ_n and oracle ψ_n, then it has to ask each query of length $2^{2^n}-1$ to correctly compute the length (otherwise we may change the value in the query of length $2^{2^n}-1$ that was not asked). Thus, for all n

$$\mathrm{time}_{M^{\psi_n}}(0^{2^{2^n}-1}) \geq 2^{2^{2^n}-1}.$$

By the definition of ψ_n it holds that $|\psi_n|(k) \leq 2^{2^{2^{n-1}}}$ for all k. Note that whenever l is monotone and bounded by $m \in \mathbb{N}$, i.e. $l(k) \leq m$ for all $k \in \mathbb{N}$, then there exists a polynomial p such that

$$P(l,k) \leq \max\{p(m), p(k)\}.$$

Therefore, it holds for all n and appropriate $C, d \in \mathbb{N}$ that

$$P(|\psi_n|, 2^{2^n}-1) \leq \max\{C2^{d2^{2^{n-1}}}, p(2^{2^n}-1)\}.$$

Using that P is a running time of $M^?$ and the inequality from above obtain

$$2^{2^{2^n}-1} \leq \mathrm{time}_{M^{\varphi_n}}(0^{2^{2^n}-1}) = \mathrm{time}_{M^{\psi_n}}(0^{2^{2^n}-1}) \leq \max\{C2^{d2^{2^{n-1}}}, p(2^{2^n}-1)\}.$$

The maximum on the far right is assumed by the first term only for finitely many n: $2^{2^n}-1 \leq d2^{2^{n-1}} + \mathrm{lb}(C)$ is a quadratic inequality for $x := 2^{2^{n-1}}$ and the set where it is fulfilled can be specified explicitly. However, this implies that the left hand side is bounded by a polynomial in $2^{2^n}-1$ which is clearly not the case. A contradiction. ◀

2.3 A partial recovery

The previous section proved that for an arbitrary set $A \subseteq \mathcal{B}$, it can not be expected that every A-restricted polynomial-time computable functional has a total polynomial-time computable total extension. However, Σ^{**} is far from being an arbitrary set. Recall the following notion:

▶ **Definition 12.** Let A be a subset of $\mathcal{B}$. A mapping $R\colon \mathcal{B} \to A$ is called a **retraction** of $\mathcal{B}$ onto A, if for all $\varphi \in A$ it holds that $R(\varphi) = \varphi$.

A property of Σ^{**} that guarantees the existence of total polynomial-time computable extensions is the following:

▶ **Lemma 13.** *There is a polynomial-time computable retraction from $\mathcal{B}$ onto Σ^{**}.*

Proof. For a string $\mathbf{a}$ let $\mathbf{a}^{\leq n}$ denote its initial segment of length n (or the string itself if it has less than n bits). Consider the mapping

$$R(\varphi)(\mathbf{a}) := \varphi(\mathbf{a})^{\leq |\varphi(0^n)|} 0^{\max\{|\varphi(0^n)| - |\varphi(\mathbf{a})|, 0\}}.$$

This mapping is a polynomial-time computable retraction from $\mathcal{B}$ onto Σ^{**}. ◀

▶ **Theorem 14.** *Whenever there is a polynomial-time computable retraction from $\mathcal{B}$ onto A, then any A-restricted polynomial-time computable functional has a total polynomial-time computable extension. I.e.* $\mathrm{P}\,|_A = \mathrm{P}(A)$.

Proof (Idea). The proof of Proposition 4 that the composition of two polynomial-time computable functionals is polynomial-time computable remains valid if the assumptions are weakened to F being $G(\mathcal{B})$-restricted polynomial-time computable. Thus, the composition of the A-restricted polynomial-time computable functional with the retraction is polynomial-time computable. ◀

The previous two results directly entail the following:

▶ **Corollary 15** ($\mathrm{P}(\Sigma^{**}) = \mathrm{P}\,|_{\Sigma^{**}}$)**.** *A functional $F : \Sigma^{**} \to \mathcal{B}$ is polynomial-time computable in the sense of Kawamura and Cook if and only if it has a total polynomial-time computable extension.*

An alternative proof can be obtained by adding a clocks to machines. (More details about how to clock machines can be found in the proof of Theorem 25.)

3 Query dependent step restrictions

This section investigates a different approach to measuring the running time of an oracle machine. The alternative approach does not rely on higher order objects as running times. Recall that for a regular Turing machine the time function $\mathrm{time}_M : \Sigma^* \to \mathbb{N}$ is defined to return on input $\mathbf{a}$ the number of steps that it takes until the machine terminates on input $\mathbf{a}$. A running time of the machine is then defined to be a function $t : \mathbb{N} \to \mathbb{N}$ such that

$$\forall \mathbf{a} \in \Sigma^* : \mathrm{time}_M(\mathbf{a}) \leq t(|\mathbf{a}|). \tag{rt}$$

For an oracle Turing machine, each of the time functions $\mathrm{time}_{M^\varphi}$ may be different. Thus, the above definition has to be replaced. The most common replacement is to replace t by a higher type object as discussed in the previous section. However, there exist other approaches of how to replace this definition in literature that stay with functions of type $\mathbb{N} \to \mathbb{N}$ for

running times of oracle machines. To distinguish these objects from the time function and the running times from second-order complexity theory, we refer to the bounds as 'step-counts' instead of 'running times'. One example of a definition in this vein has been investigated by Stephen Cook [2]. He bounds the steps an oracle Turing machine may take by modifying (rt) as follows: He replaces $|\mathbf{a}|$ by the maximum m_{M^φ} of $|\mathbf{a}|$ and the biggest length of any return value of the oracle in the run of M^φ. Then he additionally universally quantifies over $\varphi \in \mathcal{B}$. Thus ending up with

$$\forall \varphi \in \mathcal{B}, \forall \mathbf{a} \in \Sigma^* \colon \operatorname{time}_{M^\varphi}(\mathbf{a}) \leq t(m_{M^\varphi}). \tag{ot}$$

He refers to the class of functionals that can be computed by an oracle machine such that the above is fulfilled for some polynomial OPT (for 'oracle polynomial time').

3.1 Step-counts

We use a more complicated definition that turns out to be considerably more well-behaved.

▶ **Definition 16.** Let $M^?$ be an oracle Turing machine. For a given oracle φ and a given input $\mathbf{a}$ denote the content of the oracle answer tape in the k-th step of the computation by $\mathbf{b}_k$. Define the **length revision function** $o_{\varphi,\mathbf{a}} : \mathbb{N} \to \mathbb{N}$ recursively as follows:

$$o_{\varphi,\mathbf{a}}(0) := |\mathbf{a}| \quad \text{and} \quad o_{\varphi,\mathbf{a}}(n+1) := \max\{o_{\varphi,\mathbf{a}}(n), |\mathbf{b}_{n+1}|\}.$$

Note that $o_{\varphi,\mathbf{a}}(k+1) > o_{\varphi,\mathbf{a}}(k)$ means that in the k-th step of the computation, the machine asked an oracle query and the answer was bigger than both the input $\mathbf{a}$ and any of the answers the oracle has given earlier in the computation. We call this a **length revision** as it means that it became apparent to the machine that its input (the oracle) is bigger than what the previous evidence indicated.

For an oracle machine $M^?$ with a fixed oracle $\varphi \in \mathcal{B}$ let $\operatorname{time}_{M^\varphi}(\mathbf{a}) \in \mathbb{N} \cup \{\infty\}$ be the number of steps that the computation of M^φ takes on input $\mathbf{a}$. I.e. the machine is explicitly allowed to diverge on some inputs.

▶ **Definition 17** (compare Figure 1)**.** A function $t \colon \mathbb{N} \to \mathbb{N}$ is a **step-count** for an oracle Turing machine $M^?$ if

$$\forall \varphi \in \mathcal{B}, \forall \mathbf{a} \in \Sigma^*, \forall n \leq \operatorname{time}_{M^\varphi}(\mathbf{a}) \colon n \leq t(o_{\varphi,\mathbf{a}}(n)). \tag{sc}$$

Denote the set of all functionals on the Baire space that can be computed by an oracle Turing machine that has a polynomial step-count by PSC.

Note that in contrast to equations (rt) and (ot), the above is not void if the machine diverges on some inputs. The relationship between termination of a machine and the existence of a step-count is quite involved. For instance: If a machine has a step-count and diverges, then the machine queries the oracle an infinite number of times. Furthermore, if there is an integer bound on the length of all return values of an oracle, then every machine that has a step-count terminates when given that oracle and an arbitrary input.

Note that $o_{\varphi,\mathbf{a}}(\operatorname{time}_{M^\varphi}(\mathbf{a}))$ is by definition the maximum of the length of $\mathbf{a}$ and the biggest oracle query done in the computation of $M^\varphi(\mathbf{a})$. This number was previously called m_{M^φ}. Thus, Stephen Cook's class OPT (see equation (ot)) can be reproduced by not quantifying over all $n \leq \operatorname{time}_{M^\varphi}(\mathbf{a})$ but only considering the case $n = \operatorname{time}_{M^\varphi}(\mathbf{a})$. In upcoming proofs it is used that it is possible to clock a machine while basically maintaining the same step-count. This is done by checking in each step that the requirement of being a step-count is fulfilled.

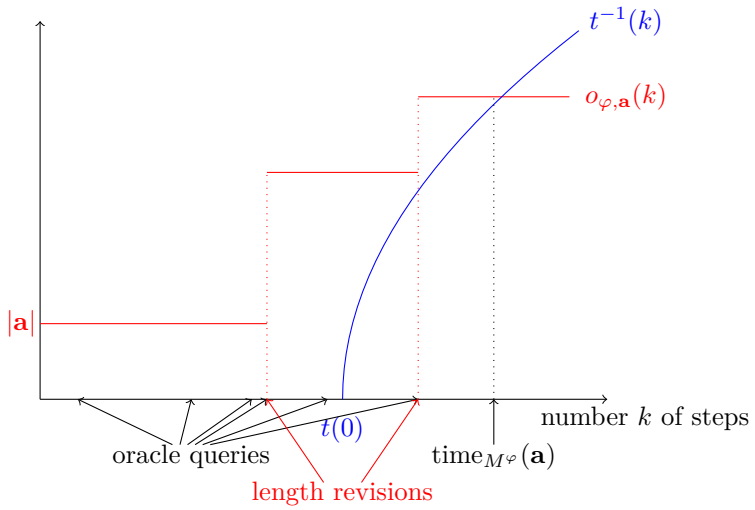

Figure 1 Verifying that φ and $\mathbf{a}$ are not a counterexample of t being a step-count. Under the assumption that t is invertible on the set $[t(0), \infty)$.

Note that this is a priory not possible for the machines used by Cook without increasing the step-count considerably. His definition allows to retroactively justify high time-consumption early in the computation by a big oracle answer late in the computation.

The very example that Cook used to disregard the class OPT as a candidate for the class of polynomial-time functionals can be used to also disregard the class of total functionals that are computed by a machine that allows a polynomial step-count:

▶ **Example 18** (PSC $\not\subseteq$ P)**.** The total functional $F : \mathcal{B} \to \mathcal{B}$ defined by

$$F(\varphi)(\mathbf{a}) := \varphi^{|\mathbf{a}|}(0)$$

can be computed by an oracle Turing machine that has a polynomial step-count but does not carry polynomial-time computable input to polynomial-time computable output.

To see that this machine has a polynomial step-count, note that it can be computed by the machine that proceeds as follows: It copies the input to the memory tape and writes 0 to the oracle query tape. Then as long as the memory tape is not empty it repeats the following steps: First copies the content of the oracle answer band to the oracle query band. Then it removes the content of the last non-empty cell from the memory band. Finally it enters the oracle query state. When the memory tape is empty it copies the content of the oracle answer band to the output tape and enters the termination state.

Copying a string of length n takes $\mathcal{O}(n)$ steps. The length of the string that has to be copied is always bounded by the previous oracle answers. The loop is carried out exactly $|\mathbf{a}|$ times. Therefore, there is some step-count from $\mathcal{O}(n^2)$.

To verify that the functional does not preserve the class of polynomial-time computable functionals consider the polynomial-time computable functional $\psi(\mathbf{a}) := \mathbf{aa}$. Note that

$$F(\psi)(\mathbf{a}) = \psi^{|\mathbf{a}|}(0) = 0^{2^{|\mathbf{a}|}}.$$

Therefore, writing $F(\psi)(\mathbf{a})$ takes at least $2^{|\mathbf{a}|}$ steps and thus $F(\psi)$ cannot be polynomial-time computable.

This means that further restrictions are necessary. In [2] this is the point where Stephen Cook decides to use polynomial-time computable functionals. This paper presents a different set of restrictions that can be used.

3.2 Finite length-revision

Let $M^?$ be an oracle Turing machine that always terminates. Then for any oracle φ and any string $\mathbf{a}$ the computation of M^φ on $\mathbf{a}$ is finite and only queries the oracle a finite number of times. Note that the number $\#o_{\varphi,\mathbf{a}}(\mathbb{N})$ of elements of the image of the the length revision function coincides with the number of length revisions that happens during the computation on oracle φ and input $\mathbf{a}$. Thus, the following statement holds true:

$$\forall \varphi \in \mathcal{B}, \forall \mathbf{a} \in \Sigma^* \colon \exists N \in \mathbb{N} \colon \#o_{\varphi,\mathbf{a}}(\mathbb{N}) \leq N.$$

In general N depends on the choice of the oracle and the string. Our restriction on the behavior of the machine is that there is an N that works independently of the choice of the oracle and the input.

▶ **Definition 19.** We say that an oracle Turing machine $M^?$ has **finite length-revision** if there is an integer N such that no matter what the oracle and the input are, no more than N length revisions happen. That is, if its length revision functions $o_{\varphi,\mathbf{a}}$ fulfill

$$\exists N \in \mathbb{N} \colon \forall \varphi \in \mathcal{B}, \forall \mathbf{a} \in \Sigma^* \colon \#o_{\varphi,\mathbf{a}}(\mathbb{N}) \leq N. \tag{flr}$$

We denote the set of all functionals on the Baire space that can be computed by machines with finite length-revision by FLR.

Finite length revision does a priori neither restrict the number of oracle questions nor the length of the oracle answers: The restriction is that there is a finite number of length revisions, that is, only a finite number of times it happens that a query is asked such that the answer is strictly bigger than the input and any earlier oracle answer.

▶ **Example 20** (P $\not\subseteq$ FLR)**.** Consider the functional

$$F : \mathcal{B} \to \mathcal{B}, \quad F(\varphi)(\mathbf{a}) := 0^{\max\{|\varphi(0^n)| \mid n \leq |\mathbf{a}|\}}.$$

The straightforward implementation asks n queries, compares their lengths and returns the maximum. This can be done in time $P(l,n) = C(n + n \cdot l(n)) + C$ for some $C \in \mathbb{N}$. However, since $|\varphi(0^n)|$ may be strictly increasing when n increases, this machine does not have finite length revision.

Indeed, no machine with finite length revision can compute F, as can be proven via contradiction as follows: Assume that there was such a machine $M^?$. Let N be a bound on the length-revisions $M^?$ does. Define an oracle such that the output of $M^\varphi(0^{N+1})$ is incorrect as follows: Let $\mathbf{a}_1$ be the first oracle query that is asked in the run of the machine $M^\varphi(0^N)$. Set $\varphi(\mathbf{a}_1) := 0^{N+1}$. Thus, a length-revision happens. Let $\mathbf{a}_2$ be the next oracle query that the machine poses. Set $\varphi(\mathbf{a}_2) := 0^{N+2}$. This means that another length revision happens. Carry on in that way until $\varphi(a_N)$ is set to 0^{2N}. After asking the query $\mathbf{a}_N$, the machine can not ask another query as we may as well set the return value to be bigger again and no further length revision is allowed.

Note that the run of the machine on 0^N is identical for any oracle that fulfills $\psi(\mathbf{a}_i) = 0^{N+i}$. Let M be the number of steps the machine M^ψ takes for any of these oracles to terminate. There are $N+1$ strings of the form 0^n for $n \leq N$. Thus, at least one of these strings is not

contained within $\mathbf{a}_1, \ldots, \mathbf{a}_N$. Let 0^m be this string. Let φ be the string function defined as follows:

$$\varphi(\mathbf{b}) = \begin{cases} 0^{N+i} & \text{if } \mathbf{b} = \mathbf{a}_i \\ 0^{M+1} & \text{if } \mathbf{b} = 0^m \\ \varepsilon & \text{otherwise.} \end{cases}$$

Obviously, the run of M^φ on 0^N coincides with the one described above. Therefore the return value can have at most M bits. Since $m \leq N$ it holds that $\left|F(\varphi)(0^N)\right| \geq |\varphi(0^m)| \geq M + 1$. Thus M^φ can on input 0^N not produce the right return value.

3.3 Strong polynomial-time computability

While neither finite length revision nor having a step-count implies termination of the machine, the combination does: We mentioned that a machine that has a step-count may only diverge with oracle φ if there is no bound on the oracle answers. This, however, is forbidden by finite length revision. Therefore, if $M^?$ is a machine that has finite length-revision and a step-count, then the computation of $M^?$ with any oracle and on any input terminates.

▶ **Definition 21.** Call a functional $F : \mathcal{B} \to \mathcal{B}$ **strongly polynomial-time computable** if there is an oracle Turing machine computing F that has both finite length-revision and a polynomial step-count (see Definition 17). We denote the set of all strongly polynomial-time computable operators by SP

As the name suggests, strong polynomial-time computability implies polynomial-time computability.

▶ **Lemma 22** (SP $\subseteq$ P)**.** *Any total strongly polynomial-time computable functional is polynomial-time computable.*

Proof. Let $M^?$ be the Turing machine that verifies that the total functional is strongly polynomial time computable, p a polynomial step-count of the machine and N a bound of the number of length revisions it does. To see that the machine runs in polynomial time fix some arbitrary oracle φ and a string $\mathbf{a}$. By the definition of being a step-count, the first oracle query in the run of M^φ on input $\mathbf{a}$ that leads to a length revision is done after at most $p(|\mathbf{a}|)$ steps and has at most this number of bits. Thus the return value of the oracle has at most length $|\varphi|(p(|\mathbf{a}|))$. Therefore, again since p is a step-count, the next oracle query that leads to a length revision can not have more than $p(|\varphi|(p(|\mathbf{a}|)))$ bits. Repeating the above argument N times and using that N is a bound of the number of length revisions proves that the computation terminates within at most $(p \circ |\varphi|)^N(p(|\mathbf{a}|))$ steps. That is, that the second order polynomial $P(l, n) := (p \circ l)^N(p(n))$ is a running time of $M^?$. ◀

Strong polynomial-time is strictly more restrictive than polynomial-time computability.

▶ **Lemma 23** (SP $\subsetneq$ P)**.** *There exists a polynomial-time computable functional that is not computable with finite length-revision. In particular, this functional is not strongly polynomial-time computable.*

Proof. An functional polynomial-time computable functional that can not be computed by a machine with finite length revision was discussed in detail in Example 20. Since SP $\subseteq$ FLR, this proves that the inclusion SP $\subseteq$ P from the previous result is strict. ◀

A candidate for a natural example of an operator from analysis that is not strongly polynomial-time computable is constructed in [1].

3.4 Compatibility with relativization

For strong polynomial-time computability, relativized notions can be introduced analogously to Section 2.1: Let $A \subseteq \mathcal{B}$. A machine $M^?$ is said to run in A-restricted strongly polynomial time if the number of length revisions $M^?$ does on oracles from A is bounded by a number and there is a polynomial step-count that is valid whenever the oracle is from A. That is if the formulas (flr) from Definition 17 and (sc) from Definition 19 are fulfilled if '$\forall \varphi \in \mathcal{B}$' is replaced by '$\forall \varphi \in A$'. Again, we denote the set of all functionals whose domain is A and that can be computed by an A-restricted strongly polynomial-time machine by $\mathrm{SP}(A)$ and the set of all functionals whose domain is contained in A and that have a total strongly polynomial-time computable extension by $\mathrm{SP}|_A$. For strong polynomial-time computability these classes coincide. This may be interpreted as strong polynomial-time computability being more well behaved with respect to partial functionals.

▶ **Lemma 24** ($\mathrm{SP}(A) = \mathrm{SP}|_A$)**.** *Any A-restricted strongly polynomial-time computable functional has a total strongly polynomial-time computable extension.*

Proof. Let $F : A \to \mathcal{B}$ be an A-restricted strongly polynomial-time computable functional and let $M^?$ be a machine that witnesses the strong polynomial-time computability of the functional. Let N be maximum number of length revisions $M^?$ does on any oracle from A and let p be a polynomial step-count valid for input from A. Define a new machine $\tilde{M}^?$ as follows: $\tilde{M}^?$ starts by initializing a counter with N written on it. Furthermore it saves the length of the input string and produces the coefficients of p on the memory tape. It applies the polynomial p to the length of the input and initializes a second counter holding this value. Now it follows the exact same steps $M^?$ does as long as no oracle query is done and meanwhile counts down the second counter. If the second counter hits zero, it terminates and returns ε. Whenever an oracle call is done, the machine checks whether the answer is longer than the answers seen before, if so, it decreases the first counter. If the counter was already zero, it terminates and returns ε. If it was not, it writes the maximum of the previous content and polynomial p applied to the length of the return value minus the number of steps taken so far to where it originally noted the length of the input. It applies the polynomial to this new value and ads the difference to the previous value to the second counter. Then it continues as before.

It is clear that the machine described above does at most $N + 1$ length revision, that it has a polynomial step-count (that depends only on p and N) and that whenever the oracle is from A, none of the counters will hit zero and $\tilde{M}^\varphi$ and M^φ produce the same values in the end. Thus $\tilde{M}^?$ computes a total strongly polynomial-time computable extension of F. ◀

3.5 Comparison to polynomial-time on Σ^{**}

Recall that originally polynomial-time computability was only defined for machines that compute total functions. Kawamura and Cook's framework for complexity of operators in analysis, however, does not require a realizer to have a total polynomial-time computable extension, but instead gives a new definition of what polynomial-time computability of a functional on Σ^{**} means. Earlier, this notion of complexity was called being Σ^{**}-restricted polynomial-time computable and the class of these functionals was denoted by $P(\Sigma^{**})$. This section proves that, for a functional whose domain is contained in Σ^{**}, the four notions of having a total extension from P or SP, or having an extension to all of Σ^{**} from $\mathrm{SP}(\Sigma^{**})$ or $P(\Sigma^{**})$ coincide. Part of this was already proven in Lemma 24, which implies that having an extension from SP and from $\mathrm{SP}(\Sigma^{**})$ are equivalent. This implies that the domain of the functional considered in Example 20 was necessarily not contained in Σ^{**}.

▶ **Theorem 25** ($\mathrm{SP}(\Sigma^{**}) = \mathrm{P}(\Sigma^{**})$)**.** *A functional is Σ^{**}-restricted polynomial-time computable if and only if it is strongly polynomial-time computable.*

Proof. That $\mathrm{SP}(\Sigma^{**}) \subseteq \mathrm{P}(\Sigma^{**})$ follows from previous results: By Lemma 24 every element of $\mathrm{SP}(\Sigma^{**})$ has a total, strongly polynomial-time computable extension. By Lemma 22, this extension is polynomial-time computable and therefore in particular Σ^{**}-restricted polynomial-time computable.

For the other direction let $M^?$ be a machine that computes some $F : \Sigma^{**} \to \mathcal{B}$ in Σ^{**}-restricted polynomial time. Define a new machine $\tilde{M}^?$ as follows: When given φ as oracle and a string $\mathbf{a}$ as input, the machine computes $P(l, m)$ with $m := |\mathbf{a}|$ and $l(n) := |\varphi(0^n)|$. This can be done by carrying out a finite sequence of oracle queries and applications of (possibly multivariate) polynomials. This sequence of operations (in particular what polynomials are applied) is determined solely from the second-order polynomial P. (More information about how to construct the polynomials can be found in Appendix A.) It writes the result into a counter, does a final query of the oracle of this length and then caries out the computations $M^?$ does on oracle φ and input $\mathbf{a}$ while counting this counter down. If the counter runs empty or an oracle answer bigger than any previous answer is encountered it terminates and returns ε. If $M^?$ terminates before this happens, it returns $M^{\varphi}(\mathbf{a})$.

While computing the value of the second-order polynomial, the number of oracle calls machine $\tilde{M}^?$ does, and therefore also the number of length revisions, is bounded by the number of applications of the length function done in the second-order polynomial P. The computation is aborted if the simulation of the machine $M^?$ leads to another length revision, thus the machine $\tilde{M}$ has finite length revision. To see that the function has a polynomial step-count note that applying polynomials from a fixed finite set of polynomials is unproblematic due to the time-constructibility of polynomials and that the simulation takes at most a number of steps that is linear in the size of the second to last oracle query before the simulation started. Thus, the machine $\tilde{M}^?$ runs in strongly polynomial time and therefore also in Σ^{**}-restricted strongly polynomial time.

Finally argue that the machine $\tilde{M}^?$ computes an extension of F: Whenever φ is length-monotone, the value $P(l, m)$ written to the counter coincides with $P(|\varphi|, |\mathbf{a}|)$. Since $M^?$ computes F in Σ^{**}-restricted time bounded by P it does not happen that the timer runs out. Due to the final oracle query $\tilde{M}^?$ does before starting to simulate $M^?$ and the length monotonicity of φ it does not happen that the simulation of $M^?$ triggers another length revision. Thus, the machine $\tilde{M}^?$ returns $M^{\varphi}(\mathbf{a})$ and therefore computes a total extension of F. ◀

4 Conclusion

The results of this paper are tightly connected to questions of whether or not it is possible to add clocks to certain machines. Clocking is a standard procedure to increase the domain of machines while maintaining its behavior on a set of 'important' oracles and inputs. For regular Turing machines, clocking allows to turn any machine that runs in polynomial time on the inputs the user cares about into a machine that actually runs in polynomial time: Take the polynomial that bounds the running time on the important inputs and in each step check if this number of steps was exceeded. This machine runs in about the same time as the original machine due to the time constructibility of polynomials. When moving to oracle Turing machines, the polynomials have to be replaced by second-order polynomials and unfortunately, these turn out not to be time constructible. Thus, for oracle Turing machines the above procedure does not extend in a straight forward manner. Indeed, Theorem 11 proves that it is impossible to clock a polynomial-time machine in general.

The framework of Kawamura and Cook and their restriction to length-monotone functions avoids the implications of this by imposing assumptions on the domains of the functionals that are considered. The notion of strong polynomial-time computability introduced in this paper tackles the same problem from another angle: For any domain it introduces a class of functionals such that clocking is possible and thus total polynomial-time computable extensions exist. Furthermore, the extensions can be chosen strongly polynomial-time computable again. However, for total functions, strong polynomial-time computability is a strictly stronger condition than polynomial-time computability.

Strong polynomial-time computability is another step towards the expectations of programmers what programs with subroutine calls should be fast. It removes dependencies of the running time on information that can not be read from the oracle in a fast way. Strong polynomial-time computability also has an intuitive meaning: A program with a subroutine that runs with length revision number N and polynomial step-count can be turned into another program with subroutine calls that computes the same output in about the same time but additionally behaves as follows: Whenever you start the program, it provides you with a time promise. Not a promise that it will terminate in the specified time, but instead a promise that within this time it will beep, revise the time promise and provide you with a reason for the revision in form of a complicated return value of a function call. Furthermore the machine will at most beep N times.

The research presented in this paper provides several starting-points for further investigations. For instance, the notion of a step-count can easily be tweaked to count the number of memory cells that are in use instead of the steps the computation takes. Such a notion could for instance be compared to the classes of operators computable with restricted space that have been introduced to the Framework of Kawamura and Cook by Kawamura and Ota [9]. This seems in particular promising as it turns out that space restricted computation in the presence of oracles is more complicated in the framework of Kawamura and Cook: To preserve the usual inclusions of complexity classes the model of computation has to be tweaked. The regular oracle Turing machines need to be replaced by machines that have a finite height stack of oracle tapes. The height of the oracle stack is another characteristic number of such a machine. Availability of an alternative point of view on space restricted computation might lead to a better understanding. Other possible follow-ups would be to investigate non-deterministic or probabilistic computation.

Acknowledgements. The authors thank an anonymous referee for the extensive feedback that lead to many improvements of the paper. The second author thanks Matthias Schröder for extended discussion on the topics of the paper.

References

1 Franz Brauße and Florian Steinberg. A minimal representation for continuous functions. `https://arxiv.org/abs/1703.10044`, 2017. Preprint.

2 Stephen A. Cook. Computational complexity of higher type functions. In *Proceedings of the International Congress of Mathematicians, Vol. I, II (Kyoto, 1990)*, pages 55–69. Math. Soc. Japan, Tokyo, 1991.

3 Hugo Férée, Walid Gomaa, and Mathieu Hoyrup. Analytical properties of resource-bounded real functionals. *J. Complexity*, 30(5):647–671, 2014. `doi:10.1016/j.jco.2014.02.008`.

4 Hugo Férée and Mathieu Hoyrup. Higher order complexity in analysis, 2013. CCA. URL: `https://hal.inria.fr/hal-00915973/document`.

5 Hugo Férée and Martin Ziegler. On the computational complexity of positive linear functionals on c[0;1], 2015. MACIS conference. URL: https://hugo.feree.fr/macis2015.pdf.

6 B. M. Kapron and S. A. Cook. A new characterization of type-2 feasibility. *SIAM J. Comput.*, 25(1):117–132, 1996. doi:10.1137/S0097539794263452.

7 Akitoshi Kawamura. *Computational Complexity in Analysis and Geometry.* PhD thesis, University of Toronto, 2011.

8 Akitoshi Kawamura and Stephen Cook. Complexity theory for operators in analysis. *ACM Trans. Comput. Theory*, 4(2):5:1–5:24, May 2012. doi:10.1145/2189778.2189780.

9 Akitoshi Kawamura and Hiroyuki Ota. Small complexity classes for computable analysis. In *Mathematical foundations of computer science 2014. Part II*, volume 8635 of *Lecture Notes in Comput. Sci.*, pages 432–444. Springer, Heidelberg, 2014. doi:10.1007/978-3-662-44465-8_37.

10 Akitoshi Kawamura and Arno Pauly. Function spaces for second-order polynomial time. In *Language, life, limits*, volume 8493 of *Lecture Notes in Comput. Sci.*, pages 245–254. Springer, Cham, 2014. doi:10.1007/978-3-319-08019-2_25.

11 Akitoshi Kawamura, Florian Steinberg, and Martin Ziegler. Complexity theory of (functions on) compact metric spaces. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS'16, pages 837–846, New York, NY, USA, 2016. ACM. doi:10.1145/2933575.2935311.

12 Akitoshi Kawamura, Florian Steinberg, and Martin Ziegler. Towards computational complexity theory on advanced function spaces in analysis. In Arnold Beckmann, Laurent Bienvenu, and Nataša Jonoska, editors, *Pursuit of the Universal: 12th Conference on Computability in Europe, CiE 2016, Paris, France, June 27 – July 1, 2016, Proceedings*, pages 142–152. Springer, Cham, 2016. doi:10.1007/978-3-319-40189-8_15.

13 Aktioshi Kawamura and Florian Steinberg. Polynomial running times for polynomial-time oracle machines. https://arxiv.org/abs/1704.01405, 2017. Preprint.

14 Branimir Lambov. The basic feasible functionals in computable analysis. *J. Complexity*, 22(6):909–917, 2006. doi:10.1016/j.jco.2006.06.005.

15 Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *J. Comput. System Sci.*, 12(2):147–178, 1976. Sixth Annual ACM Symposium on the Theory of Computing (Seattle, Wash., 1974).

16 Matthias Schröder and Florian Steinberg. Bounded time computation on metric spaces and Banach spaces. https://arxiv.org/abs/1701.02274, 2017. Preprint; extended abstract accepted for LICS 2017 conference.

A Descriptions of second-order polynomials

This appendix closes two gaps in the proofs of this paper: In the proof of Theorem 25 it was claimed that evaluating a second-order polynomial can be done by 'carrying out a finite sequence of oracle queries and applications of (possibly multivariate) polynomials' whenever queries are know that force maximal length of the return value. This appendix proves that there is such a collection of multivariate polynomials for each second-order polynomial. As a side product it provides a more elegant proof of Lemma 3 than the suggested tedious induction of the term structure.

Recall from Section 2.1 that a second-order polynomial is a function $P : \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ whose values are determined from a finite list of applications of certain rules. These rules can be understood as introduction of first-order polynomials, closure under addition, closure under multiplication and closure under function application. It seems reasonable to bundle the uses of the 'closure under addition' and the 'closure under multiplication' rules that

happen between two uses of the 'closure under function application' rule together to applying a multivariate polynomial. Formally this procedure can be described as follows:

▶ **Definition 26.** A **polynomial tree** is a finite tree T whose nodes are elements of $\mathbb{N}[X_0, \dots, X_k]$ where k coincides with the number of children the node has and there is a specified linear order on the children of each node.

Given a polynomial tree, recursively assign to each node a second-order polynomial: To a leaf t assign the second order polynomial $(l, n) \mapsto t(n)$. Now assume that second-order polynomials $P_1, \dots, P_k$ were assigned to each of the children $t_1, \dots, t_k$ of a node t. Assign to t the second-order polynomial

$$(l, n) \mapsto t(n, l(P_1(l, n)), \dots, l(P_k(l, n))) = t(n, P_1^+, \dots, P_k^+).$$

▶ **Definition 27.** A polynomial tree is called a **description** of a second-order polynomial P if P is assigned to the root of the tree by the above procedure.

Note that there may exist many different descriptions of the same second-order polynomial. For instance, both of the polynomial trees below are descriptions of the second-order polynomial $(l, n) \mapsto 2l(n)$.

Whether or not such ambiguities can completely be avoided seems to be related to whether or not the operation $P \mapsto P^+$ is injective. It is not known to the authors whether or not injectivity of this mapping holds and it is not relevant to the content of this paper. However, removing the ambiguities in descriptions would provide a normal form theorem for second-order polynomials which seems highly desirable.

▶ **Lemma 28.** *Every second-order polynomial has a description.*

Proof. For the base case note that the a description consisting of a single node $p \in \mathbb{N}[X_0]$ is a description of the second-order polynomial $(l, n) \mapsto p(n)$.

To obtain a description of the point-wise sum $P + Q$ from descriptions of P and of Q, let $t_P \in \mathbb{N}[X_0, \dots, X_k]$ be the polynomial at the root of Ps description and $t_Q \in \mathbb{N}[X_0, \dots, X_m]$ the polynomial at the root of Qs description. A description of $P + Q$ is given by merging the root of the two descriptions to a node labeled with the polynomial

$$\tilde{t}(X_0, \dots, X_{k+m+1}) := t_P(X_0, \dots, X_k) + t_Q(X_{k+1}, \dots, X_{k+m+1}).$$

For the point-wise product replace $t_P + t_Q$ in the above procedure by $t_P \cdot t_Q$.

Finally note that if P is a second-order polynomial and T a description of P then adding a single node containing the polynomial X_1 above the root of T is a description of P^+. ◀

This completes the proof of Theorem 25: Pick a description of the second-order polynomial P that is a running time of the operator. This second-order polynomial can be evaluated by going through the description, applying a regular polynomial to the length of the input for each leaf of the description, doing an oracle query for each edge of the description and evaluating a multivariate polynomial for each node of the description that has children.

Descriptions are very convenient to reason about second-order polynomials. For instance a direct proof of Lemma 3 can be given. Recall that this lemma stated that whenever P and Q are second-order polynomials, then so are the mappings

$$(l, n) \mapsto P(Q(l, \cdot), n) \quad \text{and} \quad (l, n) \mapsto P(l, Q(l, n)).$$

Proof of Lemma 3. A description of the latter can be specified by replacing each leaf p of a description of P with a description of Q where the root t of the description of Q is replaced by $p \circ t$. For the former one each edge of in a description of P has to be replaced with a description of Q (where a copy of the part of the description of P below the edge is appended to each leaf of the description of Q and there are compositions again in the roots and the leafs). ◀