Type-two polynomial-time and restricted lookahead

Bruce M. Kapron*
University of Victoria
Victoria, BC, Canada
bmkapron@uvic.ca

Florian Steinberg†
INRIA
Sophia-Antipolis, France
florian.steinberg@inria.fr

Abstract

This paper provides an alternate characterization of second-order polynomial-time computability, with the goal of making second-order complexity theory more approachable. We rely on the usual oracle machines to model programs with subroutine calls. In contrast to previous results, the use of higher-order objects as running times is avoided, either explicitly or implicitly. Instead, regular polynomials are used. This is achieved by refining the notion of oracle-poly-time computability introduced by Cook. We impose a further restriction on oracle interactions to force feasibility. Both the restriction and its purpose are very simple: it is well-known that Cook's model allows polynomial depth iteration of functional inputs with no restrictions on size, and thus does not preserve poly-time computability. To mend this we restrict the number of lookahead revisions, that is the number of times a query whose size exceeds that of any previous query may be asked. We prove that this leads to a class of feasible functionals and that all feasible problems can be solved within this class if one is allowed to separate a task into efficiently solvable subtasks. Formally, the closure of our class under lambda-abstraction and application are the basic feasible functionals. We also revisit the very similar class of strongly poly-time computable operators previously introduced by Kawamura and Steinberg. We prove it to be strictly included in our class and, somewhat surprisingly, to have the same closure property. This is due to the nature of the limited recursion operator: it is not strongly poly-time but decomposes into two such operations and lies in our class.

Keywords higher-order computability, feasibility of functionals, oracle Turing machines, applied lambda-calculus

*Research partially supported by an NSERC Discovery Grant.
†Partially funded by the ANR project *FastRelax*(ANR-14-CE25-0018-01) of the French National Agency for Research

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209124

1 Introduction

In the setting of ordinary computability theory, where computation is performed on finite objects (e.g., numbers, strings, or combinatorial objects such as graphs) there is a well-accepted notion of computational feasibility, namely polynomial-time (or poly-time) computability. The extended Church-Turing thesis codifies the convention: the intuitive notion of feasibility is captured by the formal model of computability by a poly-time Turing machine.[1] From a programming perspective this can be interpreted as a formal definition of a class of programs that should be considered fast. Of course this theory only applies to programs whose execution is determined from a finite string that is considered the input. In practice, software often relies on external libraries or features user interaction. One may address this by moving to a setting where a Turing machine acts not only on finite inputs but additionally interacts with 'infinite inputs'. This leads to the familiar **oracle Turing machine** (OTM) model, where infinitary inputs are presented via an oracle that can be fed with and will return finite strings, so that only finite information about the oracle function is available at any step of the computation. The word oracle is used as no assumptions about the process that produces values are made. In particular, the oracle provides return values instantly. From the software point of view this means judging the speed of a program independently of the quality of libraries or lazy users. Since the oracle can be understood as type-one input and oracle machines to compute type-two functions, the investigation of resource consumption in this model is called second-order complexity theory.

Can a sensible account of feasible computation be given in this model? If so, can it be kept consistent with the familiar notion of poly-time for ordinary Turing machine computation and the more traditional way of using oracle machines with `0`-`1` valued oracles [6]? These problems were first posed by Constable in 1973 [5]. He proposed only potential solutions and the task was taken up again by Mehlhorn in 1976, who gave a fully-formulated model [20]. This model is centered around Cobham's scheme-based approach to characterizing poly-time [4]. While such scheme-based approaches are very valuable from a theoretical point of view, for some applications it may be desirable to have a characterization that relies on providing bounds on resource consumption in

[1] Ignoring the possibility of quantum computers.

a machine-based model. Indeed, Mehlhorn related his formulation to the oracle machine model by proving that it satisfies the **Ritchie-Cobham property**: a functional is in his class if and only if there is an oracle machine and a bounding functional from the class such that for any inputs the machine computes the value of the functional and has a running time bounded by the size of the bounding functional. The impredicative nature of Mehlhorn's OTM characterization left open the possibility of a characterisation based more closely on the type-one model of poly-time Turing machines.

Only in 1996 did Kapron and Cook show that it is possible to give such a characterisation by relying on the notions of **function length** and **second-order polynomials** [14]. The resulting class of **basic poly-time functionals** was proved equal to Mehlhorn's, providing evidence of its naturalness and opening the way for applications in diverse areas. A representative but by no means exhaustive list includes work in computable analysis [16], programming language theory [10], NP search problems [1], and descriptive set theory [28]. The model was also used as a starting point for understanding how complexity impacts classical results on computability in higher types [3, 23, 24].

Ideas similar to those used by Kapron and Cook were used for a number of logical characterizations of the basic poly-time functionals supporting appropriateness of the class. Works using logics based on bounded arithmetic [12, 27] rely on implicit representations of second-order polynomials.[2] A drawback of the Kapron-Cook approach is that length functions and second-order polynomials are not particularly natural objects to work with. For instance, the length of a function — which can be viewed as the most basic example of a second-order polynomial — is not feasible. This has direct implications for applications. The most common approach to avoid technical difficulties is to restrict to length-monotone oracles [16, 28]. This corresponds to using only a fragment of second-order complexity theory and may in turn lead to technical difficulties.

Additional support for Mehlhorn's class and insight into its structure came from initial doubts of whether it is broad enough to include all type-two functionals that should be considered feasible. Cook formulated a notion of **intuitive feasibility**, and pointed out that a type-two **well quasi-ordering functional**, which meets the criteria of intuitive feasibility, is not in Mehlhorn's class [7]. Subsequent work uncovered a number of shortcommings of the notion of intuitive feasibility. Seth provided a class satisfying the conditions but having no recursive presentation [25] and also proved that Cook's functional does not preserve the Kalmar elementary functions [26]. Attempts by Seth and later by Pezzoli to formulate further restrictions on intuitive feasibility to avoid noted pitfalls lead back to Mehlhorn's class [22].

[2]We note that a very different logical approach is provided in [19]

Cook's intuitive feasibility uses the notion of **oracle poly-time**, which is formulated using ordinary polynomials. A POTM (for 'polynomial oracle Turing machine') is an oracle machine whose running time is bounded in the maximum size of its string input and all answers returned by its oracle input during its computation on these inputs. By itself, this notion is too weak to provide a class of feasible functionals: it is well known that iterating a poly-time function may result in exponential growth and that this is possible within this class. While Cook's approach was to rule out this behaviour on a semantic level, an alternate approach explored by a number of works involves the introduction of further restrictions to the POTM model [13, 22, 25]. Most of these restrictions are fairly elaborate and in some sense implicitly use bounding by second-order polynomials.

This paper investigates less elaborate ways to restrict the behaviour of POTMs. We present two simple syntactic restrictions to this model that give proper subclasses of Mehlhorn's class and prove them to — when closed in a natural way — lead back to the familiar class of feasible functionals.

The first restriction, originally introduced by Kawamura and Steinberg, is called **finite length revision** and operators computable by a POTM with finite length revision are called **strongly poly-time computable** [18]. It is known that this excludes very simple examples of poly-time computable operators. The second restriction is similar, original to this work and we dub it **finite lookahead revision**. We call operators that are computable by such a POTM **moderately poly-time computable**. The name is motivated by our results that this class includes the strongly poly-time computable operators (Proposition 3.8) and is contained in the poly-time operators (Proposition 3.7). These inclusions are proven to be strict (Example 3.9), but in contrast to strong poly-time it requires some effort to find something that is poly-time but not moderately poly-time. Along the way we prove that in our setting an additional restriction on the POTMs that Kawamura and Steinberg impose is not actually a restriction (Lemma 3.2).

In both cases, the failure to capture feasibility is due to a lack of closure under composition. The main result of this paper (Theorem 4.8) is that each of these classes, when closed under lambda-abstraction and application, results in exactly the poly-time functionals. To prove this we establish moderate poly-time computability of limited recursion (Lemma 4.4) and provide a factorization of any moderately poly-time computable operator into a composition of two strongly poly-time computable operators (Theorem 4.9). The proof of the later turns out to have a nice interpretation: the outer operator executes the original machine while throwing exeptions in certain cases and the inner operator is an exception handler whose form only depends on restricted information about the original operator. Finally, we point out a case where composition does not lead to a loss of moderate poly-time computability (Lemma 4.10).

The notion of a POTM is what a person familiar with complexity theory would probably come up with first if asked what programs with subroutine calls should be considered efficient. The inadequacy of this model is very easy to grasp: even if the subroutine is poly-time, there is no guarantee that the combined program runs in poly-time. The two conditions of finite length and lookahead revision are very straightforward attempts to solve this issue. We prove that imposing either in addition to the POTM condition leads to feasible programs and that it remains possible to produce solutions to all feasible problems as long as one is willing to separate the task at hand into subtasks if necessary. We provide some – but far from complete – insight into when such a split is necessary and how it can be done.

A full version of this paper is available at [15].

1.1 Preliminaries

Let Σ denote any finite alphabet, and Σ^* the set of finite strings over Σ. Usually $\Sigma = \{0, 1\}$, occasionally we use a separator symbol #. The empty string is denoted ϵ, and arbitrary elements of Σ^* are denoted $\mathbf{a}, \mathbf{b}, \ldots$. If $\mathbf{a}, \mathbf{b} \in \Sigma^*$, we write $\mathbf{ab}$ to denote their concatentation, $|\mathbf{a}|$ to denote the length of $\mathbf{a}$, $\mathbf{a}^{\leq n}$ to denote $\mathbf{a}$ truncated to its n high-order bits, for $n \geq 0$. We write $\mathbf{b} \subseteq \mathbf{a}$ to indicate that $\mathbf{b}$ is an initial segment of $\mathbf{a}$ (i.e. for some $0 \leq n \leq |\mathbf{a}|$, $\mathbf{b} = \mathbf{a}^{\leq n}$). For every $k \in \mathbb{N}$ and $1 \leq i \leq k$ we note that there exist poly-time functions $\langle \cdot, \ldots, \cdot \rangle \colon (\Sigma^*)^k \to \Sigma^*$ and $\pi_{i,k} \colon \Sigma^* \to \Sigma^*$ such that $\pi_{i,k}(\langle \mathbf{a}_1, \ldots, \mathbf{a}_k \rangle) = \mathbf{a}_i$. We assume that for every k there are constants c_1, c_2 such that $|\langle \mathbf{a}_1, \ldots, \mathbf{a}_k \rangle| \leq c_1 \cdot (|\mathbf{a}_1| + \cdots + |\mathbf{a}_k|) + c_2$ and that increasing the size of any of the strings $\mathbf{a}_i$ does not decrease the size of the tuple. Tupling is lifted to functions $\varphi_1, \ldots, \varphi_k \colon \Sigma^* \to \Sigma^*$ via $\langle \varphi_1, \ldots, \varphi_k \rangle(\mathbf{a}) := \langle \varphi_1(\mathbf{a}), \ldots, \varphi_k(\mathbf{a}) \rangle$. A type 0 functional is an element of Σ^*, and for $t \in \mathbb{N}$, a type $t+1$ funtional is a mapping from functionals of type $\leq t$ to Σ^*. This paper is mostly concerned with type t functionals for $t \leq 2$.

2 Second-order complexity theory

In [14], Kapron and Cook introduce a computational model for type-two polynomial time functionals using oracle Turing machines. We begin by reviewing their model. For notational simplicity, we do this in the operator setting: Denote by $\mathcal{B} := \Sigma^* \to \Sigma^*$ the Baire space, that is the collection of all univariate type 1 functions. The elements of $\mathcal{B}$ are denoted by $\varphi, \psi, \ldots$. An operator is a mapping $F : \mathcal{B} \to \mathcal{B}$.

An **oracle Turing machine** (OTM) or for short **oracle machine** is a Turing machine that has distinguished and distinct **query** and **answers** tapes and a designated **oracle state**. The run of an oracle machine M on oracle φ and input $\mathbf{a}$ proceeds as the run of a regular machine on input $\mathbf{a}$, but whenever the oracle machine enters the oracle state, with $\mathbf{b}$ written on the query tape, $\varphi(\mathbf{b})$ is placed immediately on the answer tape, and the read/write head returns to its initial position on both of these tapes. If the machine terminates we denote the result by $M^{\varphi}(\mathbf{a})$.

The number $\mathrm{time}_M(\varphi, \mathbf{a})$ of steps an oracle machine M takes given oracle φ and input $\mathbf{a}$ is counted as in a Turing machine with the following addition already implied above: entering the oracle state takes one time step, but there is no cost for receiving an answer from the oracle. [3] The running time of an oracle machine usually depends on the oracle.

2.1 Second-order time bounds and poly-time

To be able to talk about bounds for this running time it is necessary to have a notion for the size of an oracle.

Definition 2.1. For a given oracle function $\varphi \colon \Sigma^* \to \Sigma^*$ define its **size function** $|\varphi| \colon \mathbb{N} \to \mathbb{N}$ by

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

This suggests the type $\mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ as the right type for running times: if T is a function of this type, we say that the running time of an oracle machine M is bounded by T if for all oracles $\varphi : \Sigma^* \to \Sigma^*$ and all strings $\mathbf{a}$ it holds that

$$\mathrm{time}_M(\varphi, \mathbf{a}) \leq T(|\varphi|, |\mathbf{a}|).$$

The only thing left to do is to pick out the time bounds that should be considered polynomial.

Definition 2.2. The set of **second-order polynomials** is the smallest subset of $\mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ that contains the functions $(l, n) \mapsto 0$, $(l, n) \mapsto 1$, $(l, n) \mapsto n$, is closed under pointwise addition and multiplication and such that whenever P is an element, then so is $(l, n) \mapsto l(P(l, n))$.

We may now use Kapron and Cook's characterization [14] as our definition of Mehlhorn's class:

Definition 2.3. An operator $F \colon \mathcal{B} \to \mathcal{B}$ is **polynomial-time computable** or for short **poly-time** if there is an oracle machine M and a second-order polynomial P such that for all oracles φ and all strings $\mathbf{a}$ it holds that

1. $F(\varphi, \mathbf{a}) = M^{\varphi}(\mathbf{a})$
2. $\mathrm{time}_M(\varphi, \mathbf{a}) \leq P(|\varphi|, |\mathbf{a}|)$

We use $\mathbf{P}$ to denote the class of all poly-time operators.

This notion gives rise to a notion of poly-time computable functionals of type two. By abuse of notation we also refer to this class of functionals by $\mathbf{P}$. The functional view becomes important in Section 4.

Remark In literature the above notion is often referred to as 'basic poly-time'. As discussed in the introduction this is due to past uncertainties about the class being broad enough. We believe that enough evidence has been gathered that the class is appropriate and therefore drop the 'basic'.

[3] In this paper, we follow the *unit-cost* model [20], as opposed to the *length-cost* model [14]. Note that while an answer is received from the oracle in a single step, any further processing takes time dependant on its length.

Consider the following result taken from [18] that implies the closure of poly-time computable operators under composition.

Theorem 2.4. *Let P, Q be second-order polynomials that bound the running times of two oracle machines. Then there exists an oracle machine K that computes the composition of the corresponding operators and a $C \in \mathbb{N}$ such that for all φ and* $\mathbf{a}$

$$\mathrm{time}_K(\varphi, \mathbf{a}) \leq C \cdot P(|\varphi|, Q(P(|\varphi|, \cdot), |\mathbf{a}|))Q(P(|\varphi|, \cdot), |\mathbf{a}|) + 1).$$

The proof is straightforward and the reader not familiar with the setting may sketch a proof to get a feeling for oracle machines, running time bounds and second-order polynomials. Unsurprisingly, the machine K is constructed by replacing the oracle query commands in the program of the machine computing the outer operator by copies of the program of the machine computing the inner operator and slightly adjusting the rest of the code. Note how this result lends itself to generalizations: Kawamura and Steinberg use it to lift closure under composition to a class of partial operators they still refer to as poly-time computable. The proof also remains valid if the second-order polynomials P and Q are replaced by more general functions S and T that fulfill a monotonicity condition.

Reasoning about second-order polynomials as bounding functions can at times be tricky. Their structure theory is significantly less well developed than that of regular polynomials. Indeed, it is not clear whether second-order polynomials allow a nice structure theory at all. Furthermore, the use of nonfinitary objects in running times raises the question of computational difficulty of evaluating such bounds. It is a very simple task to find the length of a string from a string. In contrast, evaluating the length $|\varphi|(n) = \max_{|\mathbf{a}| \leq n} |\varphi(\mathbf{a})|$ of a string function is intuitively a hard task as it involves taking a maximum over an exponential number of inputs. The following theorem from [14] makes this intuition formal.

Theorem 2.5. *The length function is not polynomial-time computable: An operator L that fulfills*

$$|L(\varphi)(\mathbf{a})| = |\varphi|(|\mathbf{a}|)$$

cannot be poly-time computable.

As a consequence, a running time bound of an oracle machine is not very useful for estimating the time of a run on a given oracle and input. Even if the running time is a second-order polynomial P, to get the value $P(|\varphi|, |\mathbf{a}|)$ one has to evaluate the length function several times.

Of course, in this setting the task is a little silly. It is possible to evaluate the machine and just count the number of steps it takes. This results in a tighter bound that can be computed from the oracle and the input in polynomial time. However, from a point of view of clockability, the problem is relevant: given a second-order polynomial P that is interpreted as a 'budget' and an oracle machine M that need not run in polynomial time it is in general impossible to specify another machine N that runs in poly-time and such that for all oracles and inputs

$$\mathrm{time}_M(\varphi, \mathbf{a}) \leq P(|\varphi|, |\mathbf{a}|) \quad \Rightarrow \quad N^\varphi(\mathbf{a}) = M^\varphi(\mathbf{a}).$$

That is: N returns the correct value in the case that the run of M is in budget [18].

2.2 Oracle poly-time

The following notion was originally introduced by Cook [7] and has been investigated by several other authors as well [22, 25, 26]. Recall that for an oracle machine M the number of steps this machine takes on input $\mathbf{a}$ with oracle φ was denoted by $\mathrm{time}_M(\varphi, \mathbf{a})$ and that for counting the steps, the convention to count an oracle query as one step was chosen.

Definition 2.6. Let M be an oracle machine. For any oracle φ and input $\mathbf{a}$ denote by $m_{\varphi,\mathbf{a}}$ the maximum of the lengths of the input and any of the oracle answers that the machine gets in the run on input $\mathbf{a}$ with oracle φ. The machine M is said to run in **oracle poly-time** if there is a polynomial p such that for all oracles φ and inputs $\mathbf{a}$

$$\mathrm{time}_M(\varphi, \mathbf{a}) \leq p(m_{\varphi,\mathbf{a}}). \tag{sc}$$

Let **OPT** denote the class of operators that are computed by a machine that runs in oracle poly-time.

To avoid confusion with different notions of running times, we call a function $t : \mathbb{N} \to \mathbb{N}$ that fulfills the condition from (sc) in place of p a **step-count** of M. Thus, an oracle machine runs in oracle poly-time if and only if it has a polynomial step-count. An oracle machine with a polynomial step-count may be referred to as a POTM.

The nature of the restrictions imposed on POTMs significantly differs from imposing a second-order time bound as is done in Definition 2.3. Instead of using higher-order running times, the same type of function that are used as running times for regular Turing machines is used. The dependence on the oracle is accounted for by modifying the input of the function. This appears to be a relaxation of bounding by second-order polynomials. The following result of [7] shows that this is indeed the case.

Theorem 2.7 (P ⊆ OPT). *Any oracle machine that runs in time bounded by a second-order polynomial has a polynomial step-count.*

Proof. Let M be an oracle machine that runs in time bounded by a second-order polynomial P. For $n \in \mathbb{N}$ let $l_n : \mathbb{N} \to \mathbb{N}$ be the constant function with value n. The polynomial

$$p(n) := P(l_n, n)$$

is a step-count. This can be verified by replacing an arbitrary oracle with a truncated version that returns the empty string on arguments that the machine never asks for. □

It is well known that **P** forms a proper subclass of **OPT**. There exist operators in **OPT** that do repeated squaring and thus do not preserve poly-time computability.

Example 2.8 (P ⊊ OPT). The operator

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

can be computed by a machine that runs in oracle poly-time but does not preserve poly-time computability as it maps the poly-time computable function $\varphi(\mathbf{a}) := \mathbf{aa}$ to a function whose return values grow exponentially.

3 Recovering feasibility from OPT

The failure of **OPT** to preserve poly-time computability indicates that it is unsuitable as a class that captures an acceptable notion of feasibility. We may ask whether there is a natural way to restrict the POTM model to recover feasibility. One way to do this is to introduce preservation of poly-time functions as an *extrinsic* or a *semantic* restriction [21]. This is the approach taken by Cook with his notion of intuitively feasible functionals [7]. Since the formulation of Cook's restrictions is most comfortably done using lambda calculus we postpone restating them to Section 4. Here, we consider *intrinsic* or *syntactic* restrictions of the POTM model instead. While in part this is motivated by some of the drawbacks of the extrinsic approach, we believe that the syntactic approach stands on its own merit. In particular, if the syntactic condition is simple enough and checkable with minimal overhead, it provides simpler analysis techniques for showing that a particular operator is feasible.

Motivated by the difficulty encountered with repeated squaring in Example 2.8, we consider POTMs with restrictions on the oracle access that disallow this behaviour. Similar restrictions have been considered by Seth [25, 26]. Seth's class C_1 employs POTMs that are clocked with a form of dynamic bound on the size of any query made that faintly resembles second-order polynomials. Seth proves this class to coincide with Mehlhorn's class **P** and uses it to propose bigger classes. Seth also considers the class C_0 consisting of operators computable by POTMs whose number of queries to the oracle is uniformly bounded by a constant. Pezzoli considers restrictions that differ from ours in that she maximizes over all possible oracle inputs of a certain size and not only the queries that are asked by the machine [22], thereby implicitly involving the size function. Pezzoli proves her class equal to **P** and uses it to specify super-classes.

We seek conditions that disallow iteration without degenerating to C_0, whose purpose can be seen without knowledge of second-order polynomials and that do not refer to values of the functional input untouched in the computation.

3.1 Strong poly-time computability

The first restriction on **OPT** that we consider was introduced by Kawamura and Steinberg in [18]:

Definition 3.1. An oracle machine is said to run with **finite length revision** if there exists a number r such that in the run of the machine on any oracle and any input the number of times it happens that an oracle answer is bigger than the input and all of the previous oracle answers is at most r.

It should be noted that Kawamura and Steinberg use a slightly different notion of a step-count. They say that a function $t\colon \mathbb{N} \to \mathbb{N}$ is a **step-count** of an oracle machine M if for all oracles and inputs it holds that

$$\forall k \in \mathbb{N}\colon k \leq \mathrm{time}_M(\varphi, \mathbf{a}) \Rightarrow k \leq t(m_{k,\varphi,\mathbf{a}}),$$

where $m_{k,\varphi,\mathbf{a}}$ is the maximum of $|\mathbf{a}|$ and the biggest oracle answer given in the first k steps of the computation of M with oracle φ and input $\mathbf{a}$. For the function t to be a step-count in the sense of the present paper it suffices to satisfy the condition for the special choice $k := \mathrm{time}_M(\varphi, \mathbf{a})$. The reason we use the same name for both of these notions is that they are equivalent in our setting. The result is interesting in its own right as the advantage of Kawamura and Steinberg's notion is that it can be checked on the fly whether a given polynomial is a step-count of an oracle machine without the risk of spending a huge amount of time if this is not the case. We only state this for the case we are really interested in, but the proof generalizes.

Lemma 3.2. *Every oracle machine that computes an operator from* **OPT** *has a polynomial step-count (in the sense of Kawamura and Steinberg).*

Proof. The polynomial that proves a machine to be a POTM turns out to be a step-count. This can be verified by modifying an arbitrary oracle to return the empty string in all positions the machine does not look at. □

The key idea of the above proof is that the oracle can be modified arbitrarily. In a setting where not all oracles are eligible this might not be possible anymore. In this case it is advisable to work with step-counts in the sense of Kawamura and Steinberg. Since this paper only considers total operators, i.e. no restrictions are imposed on the oracles, it is irrelevant which notion is used. In particular we may formulate strong poly-time computability as introduced in [18].

Definition 3.3. An operator is **strongly poly-time computable** if it can be computed by an oracle machine that has both finite length-revision and a polynomial step-count. The class of these operators is denoted by **SPT**.

As the name suggests, strong poly-time computability implies poly-time computability. We state this as it can be deduced from the results of this paper. A direct proof is given in [18].

Proposition 3.4 (SPT ⊆ P). *The running time of a machine that has a polynomial step-count and finite length revision can be bounded by a second-order polynomial.*

Proof. This is an immediate consequence of Proposition 3.8 that proves the inclusion of **SPT** in a broader class called **MPT** that is introduced in the next section and proven to be included in **P** in Proposition 3.7. □

A merit of strong poly-time computability is that it has a direct interpretation as additional information about the running time of the machine: knowing a polynomial step-count of a program and the number r of length revisions, one can modify the program to provide real-time information about how long it estimates it will run. It can provide an estimate of the remaining computation time under the assumption that all necessary information has already been obtained from the oracle. In case new information is gained via oracle interaction it may update this estimate, but it may only do so at most r times.

A drawback of strong poly-time computability is that it severely restricts the access a machine has to certain oracles. It may always run into an increasing sequence of answers early in the computation. Once it runs out of length revisions, it can not pose any further oracle queries. There is no way to design a machine with finite length revision that does not simply abort when the revision budget is exceeded. This is reflected in the following example, first considered in [20], that shows that there are operators from **P** that are not in **SPT**.

Example 3.5 (SPT $\subsetneq$ P). The operator

$$F(\varphi)(\mathbf{a}) := 1^{\max_{\mathbf{b}\subseteq\mathbf{a}} |\varphi(\mathbf{b})|}$$

is poly-time but not strongly poly-time. Any machine computing F must query φ at every $\mathbf{b} \subseteq \mathbf{a}$. Regardless of the order in which the machine decides to ask the queries, there is always an oracle whose answers are increasing in size. However, $F \in \mathbf{P}$, as it may be computed by examining $|\mathbf{a}|$ queries, each of which is of size at most $|\varphi|(|\mathbf{a}|)$.

The idea behind the counterexample is that it is possible to construct an oracle that forces an arbitrary number of length revisions for a fixed machine and polynomial step-count. More details for a very similar example can be found in [18] and the same method is also used in Example 3.9.

3.2 Finite lookahead revision

The notion of strong poly-time computability rests on controlling the size of answers provided by calls to the oracle. While this restriction achieves the goal of disallowing anything but finite depth iteration, Example 3.5 shows that it also excludes rather simple poly-time computable operators. This suggests an alternate form of control, namely controlling the size of the queries themselves instead of the answers.

Definition 3.6. An oracle machine is said to run with **finite lookahead revision** if there exists a natural number r, such that for all possible oracles and inputs it happens at most r times that a query is posed whose size exceeds the size of all previous queries.

We are mostly interested in operators that can be computed by a machine that has both finite lookahead revision and a polynomial step-count. In keeping with the terminology of strong poly-time, we shall call the class of operators so computable **moderate poly-time**, denoted **MPT**. Consider the operator F that maximizes the size of the return value of the oracle over the initial segments of the string input. This operator was used to separate **SPT** from **P** in Example 3.5 and therefore fails to be strongly poly-time computable. The operator F is included in **MPT**: a machine computing F on inputs $\varphi, \mathbf{a}$ may just query the initial segments of $\mathbf{a}$ in decreasing order of length to obtain the maximum answer.

While the definition above seems reasonable enough, it entails that machines may need to unnecessarily pose oracle queries: to ask all interesting queries up to a certain size it may be necessary to pose a big query whose answer is of no interest to the computation just to avoid lookahead revisions during the computation. It is possible to tweak the oracle access of machines to avoid this behaviour and still capture the class of operators that are computed with finite lookahead revision. For instance one may use a finite stack of oracle tapes, where no pushing is allowed and popping requires the machine to specify a number of cells the tape is then truncated to in unary. While this model is not the most straightforward one, it is appealing since similar restrictions on oracle access have to be imposed to reason about space bounded computation in the presence of oracles [2, 17].

Proposition 3.7 (MPT $\subseteq$ P). *The running time of a machine that has a polynomial step-count and finite lookahead revision can be bounded by a second-order polynomial.*

Proof. The running time of a machine with polynomial step-count p that never does more than r lookahead revisions is bounded by the second-order polynomial

$$P(l, n) := (p \circ l)^r(p(n)) + p(n).$$

This can be proven by induction over the number of lookahead revisions. □

Proposition 3.8 (SPT $\subseteq$ MPT). *Every strongly poly-time computable operator is moderately poly-time computable.*

Proof. Let M be a machine that proves that an operator is strongly poly-time computable. Let p be a polynomial step-count of the machine. Consider the machine N that works as follows: first it checks whether $p(|\mathbf{a}|)$ is bigger than $|\mathbf{a}|$ and if so poses an oracle query of this size. Then N follows the first $p(|\mathbf{a}|)$ steps that M takes while remembering the size of the biggest oracle answer that it receives. If M terminates N returns its return value. If M encounters a length revision, then N repeats the procedure with $p(|\mathbf{a}|)$ replaced by $p(m)$, where m is the maximal size of an oracle answer that M has encountered so far. It can be checked that the number of lookahead revisions of N is no bigger than the number of length-revisions of M. Since N and M compute the same operator, this proves the assertion. □

Although **MPT** is more powerful than **SPT**, it is still not powerful enough to capture all of **P**:

Example 3.9 (MPT ⊊ P). Consider the operator $F : \mathcal{B} \to \mathcal{B}$ defined as follows: First recursively define a sequence of functions $F_i : \mathcal{B} \to \Sigma^*$ by

$$F_0(\varphi) := \epsilon \quad \text{and} \quad F_{n+1}(\varphi) := (\varphi \circ \varphi)(F_n(\varphi))^{\leq |\varphi(\epsilon)|}.$$

That is: start on value ϵ and n times iterate the process of applying the function $\varphi \circ \varphi$ and then truncating the result to have length $|\varphi|(0)$. Set

$$F(\varphi)(\mathbf{a}) := F_{|\mathbf{a}|}(\varphi).$$

On one hand this operator is poly-time: the straight forward algorithm for computing F runs in polynomial time. On the other hand it cannot be computed by a machine with finite lookahead revision and a polynomial step-count. Verifying this can be done by taking an arbitrary machine that has a polynomial step-count and computes F and for each given natural number constructing an oracle that forces the machine to make at least this number of lookahead revisions. The construction exploits the fact that a polynomial-time machine can not exhaustively search through all queries of small length and forces a sequence of increasing queries by writing the information about locations of queries with increasingly big answers at positions that can not be accessed without having gotten the previous big answer. The details of the construction can be found in the full version of this paper [15].

4 Lambda-calculi for feasible functionals

The preceding section presented two classes **SPT** and **MPT** of operators based on simple syntactic restrictions to POTMs, both of which fail to capture all of **P**. The rest of the paper proves that this is exclusively due to a failure of closure of these classes under composition.

Composition is a notion from the operator setting, but for this chapter the functional standpoint is more convenient. In the functional setting, there are more ways of combining functionals. consider for instance $F, G\colon \mathcal{B} \times \Sigma^* \to \Sigma^*$: one may apply G and hand the resulting string as input to F, i.e. send φ and $\mathbf{a}$ to $F(\varphi, G(\varphi, \mathbf{a}))$, or leave the string argument in G open and use this as function input for F: I.e. send φ and $\mathbf{a}$ to $F(\lambda\mathbf{b}.G(\varphi, \mathbf{b}), \mathbf{a})$. The latter captures the composition of operators and uses the familiar notation for lambda-abstraction. One may go one step further and also use lambda-abstraction over φ and $\mathbf{a}$ to express the two ways to combine functionals by terms in the lambda-calculus with F and G as constants. On the other hand, any term in the lambda-calculus with constants from a given class of functionals can be interpreted as a functional again. It should be noted that in general these functionals need not be type-one or two anymore as the lambda-calculus provides variables for each finite type.

This section reasons about closures of classes of functionals under λ-abstraction and application as a subsitute for closure under composition in the operator setting. We do not attempt to give a self-contained presentation of the tools from lambda-calculus needed here and point to [9] for more details. Our primary focus is on using such calculi *definitionally* – that is we are interested in the denotational semantics of type-one and -two terms, and only require operational notions to reduce arbitrary terms to such terms. In particular, we consider systems with constant symbols for every function in some type-one or -two class, without necessarily giving reduction rules for such symbols. Note that we do not initially restrict the type-level of the functionals we work with. We consider closure at all finite types and recover the classes of interest by taking sections.

Definition 4.1. For a class **X** of functionals, let $\lambda(\mathbf{X})$ denote the set of simply-typed λ-terms which include a constant symbol for every element of **X**. For a set T of terms the **one-section of** T, denoted T_1, is the class of functions represented by type-one terms of T. The **two-section of** T, denoted T_2, is the set of functionals represented by type-two terms of T.

Denote the class of poly-time computable functions by P. It is well-known that $\lambda(\mathrm{P})_1 = \mathrm{P}$. Seth proves that $\lambda(\mathrm{P})_2 = C_0$, where C_0 is his class of functionals computed by POTMs only allowed to access their oracle a finite number of times.

Mehlhorn's schematic characterization of poly-time [20] fits quite nicely into the lambda calculus approach. However, the limited recursion on notation scheme translates to a type-three constant as it produces a type-two functional from a set of type-two functionals. Work by Cook and Urquhart revealed that it is possible to use a type-two constant instead [9]. Cook and Urquart consider lambda-terms with symbols for a collection of basic poly-time computable functions as well as one type-two symbol $\mathcal{R}$ capturing limited recursion on notation. We denote the functional that Cook and Urquart use to give meaning to $\mathcal{R}$ by $\mathcal{R}'$ as for our purposes a slightly different one is more convenient. Set $\mathcal{R}'(\varphi, \mathbf{a}, \psi, \epsilon) = \mathbf{a}$ and

$$\mathcal{R}'(\varphi, \mathbf{a}, \psi, \mathbf{c}i) = \begin{cases} \mathbf{t} & \text{if } |\mathbf{t}| \leq |\psi(\mathbf{c}i)|; \\ \psi(\mathbf{c}i) & \text{otherwise.} \end{cases}$$

where $\mathbf{t} = \varphi(\mathbf{a}i, \mathcal{R}'(\varphi, \mathbf{a}, \psi, \mathbf{c}))$. Readers familiar with the Mehlhorn's limited recursion on notation scheme should note that this is an application of the scheme to feasible functionals and that $\mathcal{R}'$ itself is a feasible functional. The next section verifies this directly for a very similar functional.

Cook and Kapron consider an equivalent version of the Cook-Urquhart system which includes constant symbols for all type-one poly-time functions [8], and call the functionals that correspond to terms in this system the **basic feasible functionals**, denoted by **BFF**. This means that $\mathbf{BFF} = \lambda(\mathrm{P} \cup \{\mathcal{R}'\})$ up to identification of lambda terms with the functionals they represent. Clearly, in this setting, $\mathcal{R}'$ is needed only for higher-order recursions.

The class **BFF** contains functionals of all finite types and can be cut down to the types we are interested in by considering the sections $\mathbf{BFF}_1$ and $\mathbf{BFF}_2$. While there are reasonable doubts whether the class **BFF** captures feasiblity in all finite types [11], the significance of its two-section is demonstrated by the following result of Kapron and Cook [14].

Theorem 4.2 ($\mathbf{P} = \mathbf{BFF}_2$)**.** *The basic feasible functionals of type-two are exactly the poly-time functionals.*

It is also true that $\mathbf{BFF}_1 = \mathrm{P}$. Cook's notion of intuitively feasible functionals is based on this property of $\mathbf{BFF}_1$: a type-two functional F is **intuitively feasible** if the corresponding operator is in **OPT** and adding it to **BFF** does not change the one-section, i.e. $\lambda(\mathrm{P} \cup \{\mathcal{R}', F\})_1 = \mathrm{P}$.

4.1 Limited recursion as an operator

As mentioned, using a recursion functional slightly different from Cook and Urquart's $\mathcal{R}'$ is more convenient for our purposes. Consider the **limited recursion functional** $\mathcal{R}$, i.e. the type-two functional defined by $\mathcal{R}(\varphi, \mathbf{a}, \mathbf{b}, \epsilon) := \mathbf{a}$ and

$$\mathcal{R}(\varphi, \mathbf{a}, \mathbf{b}, \mathbf{c}i) := \varphi(\mathbf{c}i, \mathcal{R}(\varphi, \mathbf{a}, \mathbf{b}, \mathbf{c}))^{\leq |\mathbf{b}|}$$

First we establish that we may indeed swap the recursion functional $\mathcal{R}'$ with this functional.

Proposition 4.3 ($\lambda(\mathrm{P} \cup \{\mathcal{R}\}) = \mathbf{BFF}$)**.** *The recursion functionals $\mathcal{R}'$ and $\mathcal{R}$ are equivalent in the sense that*

$$\mathcal{R} \in \lambda(\mathrm{P} \cup \{\mathcal{R}'\})_2 \quad \textit{and} \quad \mathcal{R}' \in \lambda(\mathrm{P} \cup \{\mathcal{R}\})_2$$

Proof. Clearly, $\mathcal{R}(\varphi, \mathbf{a}, \mathbf{b}, \mathbf{c}) = \mathcal{R}'(\lambda\mathbf{s}\lambda\mathbf{t}.\varphi(\mathbf{s}, \mathbf{t})^{\leq |\mathbf{b}|}, \mathbf{a}, \lambda\mathbf{s}.\mathbf{b}, \mathbf{c})$. For the other direction first show that the operator

$$F(\psi)(\mathbf{c}) := \max_{\mathbf{c}' \subseteq \mathbf{c}} F(\psi)(\mathbf{c}')$$

may be defined using $\mathcal{R}$. Define F' such that $F(\psi)(\mathbf{c}) = \psi(F'(\psi, \mathbf{c}))$. Thus,

$$F'(\psi, \mathbf{c}) = \mathcal{R}(\lambda\mathbf{s}\lambda\mathbf{t}.M(\psi, \mathbf{s}, \mathbf{t}), \epsilon, \mathbf{c}, \mathbf{c}),$$

where $M(\psi, \mathbf{s}, \mathbf{t}) = \mathbf{s}$ if $\psi(\mathbf{s}) > \psi(\mathbf{t})$ and $\mathbf{t}$ otherwise. Let ℓ be the poly-time function that satisfies: $\ell(\mathbf{s}, \mathbf{t}) = \mathbf{s}$ if $|\mathbf{s}| \leq |\mathbf{t}|$ and $\mathbf{t}$ otherwise. Then

$$\mathcal{R}'(\varphi, \mathbf{a}, \psi, \mathbf{c}) = \mathcal{R}(\lambda\mathbf{s}\lambda\mathbf{t}.\ell(\varphi(\mathbf{s}, \mathbf{t}), \psi(\mathbf{s})), \mathbf{a}, F(\psi, \mathbf{c}), \mathbf{c}).$$

This proves the assertion. □

The classes **MPT** and **SPT** contain operators. To capture the limited recursion functional $\mathcal{R}$ we first need to give an equivalent operator. Indeed an arbitrary type-two functional may be translated to an operator within the lambda calculus in exactly the same way and this may be interpreted as an alternate normal form. Not surprisingly, this normal form relies on the availability of tupling functions and projections. In the situations we are most interested in these are always available, either because constants for the poly-time computable functions are available or because we can provide replacements for these.

A different way to approach this problem would be to adapt the definition of oracle machines to allow for multiple oracles and multiple input and output tapes. It is worthwhile to note that the definitions of **MPT** and **SPT** generalize in a straight forward way and that this is equivalent to using translations. The extended definition is more convenient for high level proofs as it hides all uses of tupling and projections. However, it requires additional definitions and notations and for the sake of being self-contained we mostly avoid it.

Recall that we denote the k-ary poly-time computable string-tupling functions by $\langle \cdot, \ldots, \cdot \rangle$ and that they have polynomial-time computable projections $\pi_{1,k}, \ldots \pi_{k,k}$. The **limited recursion operator** R is defined by

$$\mathrm{R}(\psi) := \lambda\mathbf{a}.\mathcal{R}(\lambda\mathbf{b}\lambda\mathbf{c}.\psi(\langle \mathbf{b}, \mathbf{c} \rangle), \pi_{1,3}(\mathbf{a}), \pi_{2,3}(\mathbf{a}), \pi_{3,3}(\mathbf{a})).$$

The right hand side is a type-two term from $\lambda(\mathrm{P} \cup \{\mathcal{R}\})$ and thus $\mathrm{R} \in \lambda(\mathrm{P} \cup \{\mathcal{R}\})_2 = \mathbf{P}$. Recall that the tupling functions for string functions were defined by $\langle \varphi, \ldots, \psi \rangle(\mathbf{a}) = \langle \varphi(\mathbf{a}), \ldots, \psi(\mathbf{a}) \rangle$. Thus, the tupling functions on string functions are also available as a type-two terms. The same is true for the projections. Therefore any type-two functional can be replaced by an operator using lambda-abstraction and application and additionally tupling functions and projections.

4.2 The lambda-closures of MPT and SPT

This section proves that $\lambda(\mathbf{MPT})_2 = \mathbf{P} = \lambda(\mathbf{SPT})_2$. For the first equality, the strategy is very simple: We prove $\mathrm{R} \in \mathbf{MPT}$. For the second equality additional work has to be done.

Lemma 4.4 ($\mathrm{R} \in \mathbf{MPT}$)**.** *The limited recursion operator is moderately poly-time computable.*

Proof. A high-level description of an oracle machine M computing R can be given as follows: On inputs ψ and $\mathbf{a}$ fix the following notations: $\mathbf{t}_0 := \pi_{1,3}(\mathbf{a})$, $\mathbf{c} := \pi_{2,3}(\mathbf{a})$ and $\mathbf{b} := \pi_{3,3}(\mathbf{a})$. The machine M operates as follows: first it queries ψ at $1^{\max\{|\langle \mathbf{c}, \mathbf{t}_0 \rangle|, |\langle \mathbf{c}, \mathbf{b} \rangle|\}}$. The return value is not used, but this query guarantees that M has exactly one lookahead revision. Then, for $i = 1, \ldots, n = |\mathbf{c}|$ set $\mathbf{t}_i \leftarrow \psi(\langle \mathbf{c}^{\leq i}, \mathbf{t}_{i-1} \rangle)^{\leq |\mathbf{b}|}$, and return $\mathbf{t}_n$. Since $|\mathbf{t}_i| \leq |\mathbf{b}|$ for $1 \leq i \leq n$, the initial query made by M is the largest. Thus M has a quadratic step-count and lookahead revision 1. □

This theorem does not rely on the reformulation of the recursion functional. It can be checked that it is also true if Cook and Urquart's formulation is used. However, the description of the machine becomes more involved.

The first main result of this section follows easily.

Theorem 4.5 ($\lambda(\mathbf{MPT}) = \mathbf{BFF}$)**.** *The functionals represented by lambda terms with symbols for moderate poly-time computable operators are exactly the basic feasible functionals.*

Proof. For any poly-time computable function $\psi \in \mathcal{B}$, the constant operator defined by $K_\psi(\varphi) := \psi$ is moderately poly-time computable. Thus the lambda-term $\lambda\mathbf{a}.K_\psi(\lambda\mathbf{b}.\mathbf{b})(\mathbf{a})$ may be used as replacement for ψ. For multiple arguments note

that the operator defined by $T(\varphi)(\mathbf{a}) := \langle \varphi(\epsilon), \mathbf{a}\rangle$ is moderate poly-time computable.

Due to the moderate poly-time computability of the limited recursion operator R and the availability of tupling functions from the first part of the proof, the lambda-term

$$\lambda\varphi\lambda\mathbf{a}\lambda\mathbf{b}\lambda\mathbf{c}.\mathrm{R}(\lambda\mathbf{d}.\varphi(\pi_1(\mathbf{d}),\pi_2(\mathbf{d})))(\langle\mathbf{a},\mathbf{b},\mathbf{c}\rangle)$$

may be used to replace the symbol $\mathcal{R}$. □

Unfortunately the same tactic is bound to fail for the strongly poly-time computable operators: an argument similar to that given for the maximization operator in Example 3.5 shows that the limited recursion operator is not in **SPT**. This forces us to attempt to split the functional into simpler parts. Due to the concrete form of our limited recursion functional, this can fairly easily be done. It should be noted though, that this is a very special case of a more general theorem proved in the next section and that understanding the decomposition in the next lemma is not crucial for the understanding of the paper and thus the proof is omitted.

Lemma 4.6 ($\mathcal{R} \in \lambda(\mathbf{SPT})_2$)**.** *There exists a lambda-term with two constants from* **SPT** *that evaluates to the limited recursion functional* $\mathcal{R}$.

The following theorem can now be proven completely analogous to the same statement for **MPT**.

Theorem 4.7 ($\lambda(\mathbf{SPT}) = \mathbf{BFF}$)**.** *The functionals represented by lambda terms with symbols for strongly poly-time computable operators are exactly the basic feasible functionals.*

Proof. The only adjustment that has to be made is to change the lambda term replacing $\mathcal{R}$ to the term over **SPT** that exists according to Lemma 4.6. □

As we are mainly interested in the two sections, we gather these special cases from the Theorems 4.5 and Theorem 4.7 and state them as the main results of this paper.

Theorem 4.8 ($\lambda(\mathbf{MPT})_2 = \mathbf{P} = \lambda(\mathbf{SPT})_2$)**.** *The type-two functionals represented by terms in the closure of both* **SPT** *and* **MPT** *under lambda-abstraction and application are exactly the poly-time computable functionals.*

4.3 Some results about composition

The decomposition of the limited recursion functional into a lambda-term over two strongly poly-time computable functionals from Lemma 4.6 is a special case of a general phenomenon. It is instructive to revisit the maximization functional from Example 3.5 with this in mind. For the statement of the corresponding result we switch back to the operator setting and provide a decomposition of any moderate poly-time computable operator into a composition of two strongly poly-time computable operators. Note that composition of operators may be expressed as a lambda term: $S \circ T = \lambda\varphi.(S \circ T)(\varphi) = \lambda\varphi.S(T(\varphi))$. Furthermore, the translations between functionals and operators were shown to be possible within the lambda-calculus over **SPT**. Thus, the decomposition from Lemma 4.6 is a special case of the following theorem.

Theorem 4.9 ($\mathbf{MPT} \subseteq \mathbf{SPT} \circ \mathbf{SPT}$)**.** *Any moderate polynomial-time computable operator can be written as composition of two strongy poly-time computable operators.*

Proof. Factor a machine M into machines $\tilde{M}$ and N that are defined as follows: The machine $\tilde{M}$ on oracle φ and on input $\langle\mathbf{a},\mathbf{b}\rangle$ first checks if the string $\mathbf{b}$ is of the form $\mathbf{c}\#\mathbf{c}'$. If it is not it returns the empty string. If it is, then it asks the query $\mathbf{c}$. The answer to that first query is ignored and instead the steps M does on input $\mathbf{a}$ and with oracle φ are carried out while keeping track of the size m of the biggest oracle answer. In case a length revision is encountered on oracle query $\mathbf{d}$, the machine checks if $\mathbf{d}$ is shorter than the string $\mathbf{c}'$. If so, $\tilde{M}$ returns $\mathbf{d}\#1^{|\mathbf{c}'|-|\mathbf{d}|}$. If not, it returns $\mathbf{d}\#\#1^m$. If M terminates it returns $M(\mathbf{a})$.

To describe how the machine N works, let p be a polynomial step-count of the original machine and r its number of lookahead revisions. With oracle ψ and on input $\mathbf{a}$ the machine N evaluates $m := p(|\mathbf{a}|)$ and poses the query $\langle\mathbf{a},\#1^m\rangle$ to the oracle. If the answer contains a # and triggers a length revision, it returns the empty string. If the answer contains a single # and does not trigger a length revision, it copies the string $\mathbf{d}$ of digits that occur before the # and poses as next query $\langle\mathbf{a},\mathbf{d}\#1^m\rangle$. If the answer contains a double #, it checks whether the length k of the string after the double # is bigger than $|\mathbf{a}|$ and if so replaces m by $p(k)$. If the answer does not contain a #, it returns the answer. If it has to repeat the procedure more than $2r+1$ times, it returns ϵ. □

To illustrate this decomposition, we informally describe its effect on the limited recursion functional $\mathcal{R}$ from Section 4.1:

$$\mathcal{R}(\varphi,\mathbf{a},\mathbf{b},\mathbf{c}) = \mathcal{P}(\lambda\mathbf{d}\lambda\mathbf{t}.\mathcal{Q}(\varphi,\mathbf{t},\mathbf{b},\mathbf{c},\mathbf{d}),\mathbf{a},\mathbf{b},\mathbf{c}),$$

where $\mathcal{Q}$ acts like a re-entrant version of $\mathcal{R}$ which may be started at an arbitrary point in the recursion, and raises an exception whenever it encounters more than one length revision. $\mathcal{P}$ acts as an exception handler that restarts $\mathcal{Q}$ in case an exception is thrown. The following is a high-level description of an oracle machine M computing $\mathcal{Q}$. On inputs $\varphi,\mathbf{t},\mathbf{b},\mathbf{c},\mathbf{d}$, first check that $\mathbf{d} \subseteq \mathbf{c}$. If not, return an "abort" value, say ϵ. Otherwise, letting $\mathbf{t}_0 = \mathbf{t}$, M does the following for $i = 1,\ldots,n = |\mathbf{c}| - |\mathbf{d}|$:

1. Set $\mathbf{c}_i \leftarrow \mathbf{c}^{\leq(i+|\mathbf{d}|)}$, $\mathbf{s}_i \leftarrow \varphi(\mathbf{c}_i,\mathbf{t}_{i-1})$ and $\mathbf{t}_i \leftarrow \mathbf{s}_i^{\leq|\mathbf{b}|}$
2. If $i > 1$ and $|\mathbf{s}_i| > |\mathbf{s}_{i-1}|$ return $\mathbf{c}_i,\mathbf{t}_i$ encoded as a string of a length only dependent on $|\mathbf{b}|$.

If all steps execute, return $\mathbf{t}_n$ marked as return value. In this case it is simple to guarantee that all exception messages have the same length and this makes it possible for the exception handler $\mathcal{P}$ to avoid length revisions. Now an OTM N computing, $\mathcal{P}$, on inputs $\psi,\mathbf{a},\mathbf{b},\mathbf{c}$, repeatedly calls ψ, starting with inputs $\epsilon,\mathbf{a}$. If any answer is marked as a return value

then N returns this answer. If an answer is an exception message, N decodes it and feeds it back to ψ. If an answer is neither, or if more than two length revisions occur or more than $|\mathbf{c}|$ iterations pass, then N aborts and returns ϵ.

Methods similar to those employed in the proof of Theorem 4.9 can be used to prove that in special cases composition does not lead outside of **MPT**.

Lemma 4.10 ($\mathbf{MPT}_1 \circ \mathbf{MPT} \subseteq \mathbf{MPT}$). *Let $F, G \in \mathbf{MPT}$ be such that the machine computing F does only one lookahead revision. Then $F \circ G \in \mathbf{MPT}$.*

5 Conclusions and future work

We have given two characterisations of feasible type-two computation which coincide with the familiar notion of poly-time, but have a simple and appealing syntactic description. They use POTMs with simple restrictions on how oracles may be accessed as building blocks. Such machines may call other POTMs as subroutines, but as long as all the machines obey the query restriction, the result is poly-time. Although we do not consider it the main contribution — the evidence is overwhelming already — these characterizations further support the naturalness of the notion of feasibility in second-order complexity theory: both of these models, formulated without any notion of length-functions, second-order polynomials, or limited recursion on notation, lead to the familiar notion. The simplicity of the characterisation should make it easier to reason about feasibility in the type-two setting.

While the results of this paper are satisfactory, they do raise a lot of additional questions. For instance we conclude that $\mathbf{SPT} \subsetneq \mathbf{MPT} \subseteq \mathbf{SPT} \circ \mathbf{SPT} \subseteq \mathbf{P}$ and that at least one more inclusion must be strict (as $\mathbf{MPT} \subsetneq \mathbf{P}$). We tried to prove the equality of $\mathbf{MPT} \circ \mathbf{MPT}$ and $\mathbf{P}$ early in our search for closure properties of **MPT**. This lead us to ideas very similar to those pursued by Seth [26]. We now believe $\mathbf{MPT} \circ \mathbf{MPT}$ and $\mathbf{P}$ to be different and similar inclusions combining more operators from **SPT** or **MPT** to be strict, but have not yet produced counterexamples. For instance the functional from Example 3.9 can be written as $F(\varphi, \mathbf{a}) = \mathcal{R}(\lambda \mathbf{b}.\varphi(\varphi(\mathbf{b})), \epsilon, \varphi(\epsilon), \mathbf{a})$. Both the limited recursor $\mathcal{R}$ and $\lambda \mathbf{b}.\varphi(\varphi(\mathbf{b}))$ are from **MPT**.

This leads to another goal, namely a more direct proof of the closure results that may provide a concrete decomposition into few elements of **SPT** or **MPT**. The number of components needed may provide a measure for the complexity of a task that resolves finer than poly-time. This is in particular interesting as there does not exist a second-order substitute for the degree of a polynomial or even linearity.

References

[1] P. Beame, S.A. Cook, J. Edmonds, R. Impagliazzo, and T. Pitassi. 1998. The Relative Complexity of NP Search Problems. *J. Comp. Sys. Sci.* 57, 1 (1998), 3–19.

[2] J. Buss. 1988. Relativized alternation and space-bounded computation. *J. Comp. Sys. Sci.* 36, 3 (1988), 351–378. Structure in Complexity Theory Conference (Berkeley, CA, 1986).

[3] S.R. Buss and B.M. Kapron. 2002. Resource-bounded continuity and sequentiality for type-two functionals. *ACM Trans. Comp. Log.* 3, 3 (2002), 402–417.

[4] A. Cobham. 1965. The intrinsic computational difficulty of functions. In *Logic, Methodology and Philosophy of Science: Proc. 1964 Intl. Congress (Studies in Logic and the Foundations of Mathematics)*, Yehoshua Bar-Hillel (Ed.). North-Holland Publishing, 24–30.

[5] R.L. Constable. 1973. Type Two Computational Complexity. In *5th Annual ACM STOC, (Austin, TX 1973)*. 108–121.

[6] S.A. Cook. 1971. The complexity of theorem-proving procedures. In *3rd Annual ACM STOC, 1971, Shaker Heights, OH*. 151–158.

[7] S.A. Cook. 1992. Computability and complexity of higher type functions. In *Logic from computer science (Berkeley, CA, 1989)*. Math. Sci. Res. Inst. Publ., Vol. 21. Springer, New York, 51–72.

[8] S.A. Cook and B.M. Kapron. 1990. Characterizations of the basic feasible functionals of finite type. In *Feasible mathematics (Ithaca, NY, 1989)*. Birkhäuser, 71–96.

[9] S.A. Cook and A. Urquhart. 1993. Functional interpretations of feasibly constructive arithmetic. *Ann. Pure Appl. Logic* 63, 2 (1993), 103–200.

[10] N. Danner and J.S. Royer. 2007. Adventures in time and space. *Log. Methods Comput. Sci.* 3, 1 (2007), 1:9, 53.

[11] H. Férée. 2017. Game semantics approach to higher-order complexity. *J. Comput. System Sci.* 87 (2017), 1–15.

[12] A. Ignjatovic and A. Sharma. 2004. Some applications of logic to feasibility in higher types. *ACM TOCL* 5, 2 (2004), 332–350.

[13] R.J. Irwin, J.S. Royer, and B.M. Kapron. 2001. On characterizations of the basic feasible functionals. I. *J. Funct. Prog.* 11, 1 (2001), 117–153.

[14] B.M. Kapron and S.A. Cook. 1996. A new characterization of type-2 feasibility. *SIAM J. Comput.* 25, 1 (1996), 117–132.

[15] B.M. Kapron and F. Steinberg. 2018. Type-two polynomial-time and restricted lookahead. preprint. (2018). https://arxiv.org/abs/1801.07485

[16] A. Kawamura and S.A. Cook. 2012. Complexity Theory for Operators in Analysis. *TOCT* 4, 2 (2012), 5:1–5:24.

[17] A. Kawamura and H. Ota. 2014. Small complexity classes for computable analysis. In *Mathematical foundations of computer science 2014. Part II*. LNCS, Vol. 8635. Springer, Heidelberg, 432–444.

[18] A. Kawamura and F. Steinberg. 2017. Polynomial running times for polynomial-time oracle machines. In *2nd International Conference on Formal Structures for Computation and Deduction, 2017, Oxford, UK*. 23:1–23:18.

[19] D. Leivant. 2002. Implicit computational complexity for higher type functionals. In *CSL*. LNCS, Vol. 2471. Springer, Berlin, 367–381.

[20] K. Mehlhorn. 1976. Polynomial and abstract subrecursive classes. *J. Comp. Sys. Sci.* 12, 2 (1976), 147–178.

[21] C.H. Papadimitriou. 1994. *Computational complexity.* Addison-Wesley Publishing Company, Reading, MA. xvi+523 pages.

[22] E. Pezzoli. 1998. On the computational complexity of type 2 functionals. In *Computer science logic (Aarhus, 1997)*. LNCS, Vol. 1414. Springer, Berlin, 373–388.

[23] J.S. Royer. 1997. Semantics vs syntax vs computations: machine models for type-2 polynomial-time bounded functionals. *J. Comp. Sys. Sci.* 54, 3 (1997), 424–436.

[24] J.S. Royer. 2004. On the computational complexity of Longley's H functional. *Theoret. Comp. Sci.* 318, 1-2 (2004), 225–241.

[25] A. Seth. 1992. There is no recursive axiomatization for feasible functionals of type 2. In *7th Annual IEEE Symposium on Logic in Computer Science (Santa Cruz, CA, 1992)*. 286–295.

[26] A. Seth. 1993. Some desirable conditions for feasible functionals of type 2. In *8th Annual IEEE Symposium on Logic in Computer Science (Montreal, PQ, 1993)*. 320–331.

[27] T. Strahm. 2004. A proof-theoretic characterization of the basic feasible functionals. *Theoret. Comput. Sci.* 329, 1-3 (2004), 159–176.

[28] M. Townsend. 1990. Complexity for type-2 relations. *Notre Dame J. Formal Logic* 31, 2 (1990), 241–262.