1997

Semantics vs. Syntax vs. Computations: Machine models for type-2 polynomial-time bounded functionals

James S. Royer
Syracuse University, School of Computer and Information Science, royer@top.cis.syr.edu

Semantics vs. Syntax vs. Computations

Machine Models for Type-2 Polynomial-Time Bounded Functionals

Intermediate Draft, Revision $2 + \epsilon$

James S. Royer

School of Computer and Information Science
Syracuse University
Syracuse, NY 13244 USA

Email: royer@top.cis.syr.edu

Abstract

This paper investigates analogs of the Kreisel-Lacombe-Shoenfield Theorem in the context of the type-2 basic feasible functionals.

We develop a direct, polynomial-time analog of effective operation in which the time bounding on computations is modeled after Kapron and Cook's scheme for their basic polynomial-time functionals. We show that if P $=$ NP, these polynomial-time effective operations are strictly more powerful on $\mathcal{R}$ (the class of recursive functions) than the basic feasible functions.

We also consider a weaker notion of polynomial-time effective operation where the machines computing these functionals have access to the computations of their procedural parameter, but not to its program text. For this version of polynomial-time effective operations, the analog of the Kreisel-Lacombe-Shoenfield is shown to hold—their power matches that of the basic feasible functionals on $\mathcal{R}$.

October 22, 1996

1. The Problem and Its Classical Solutions in Recursion Theory

In programming languages, a *higher-order procedure* is a procedure that takes as arguments, or produces as results, other procedures. Higher-order procedures are powerful programming tools and are a stock feature of many contemporary programming languages, e.g., ML, Scheme, and Haskell.

One view of a procedure is as a syntactic object. Thus, one way of reading the above definition is as follows. A higher-order procedure is a procedure that takes syntactic objects as inputs, some of which it treats as procedures and others as non-procedural data items. The value produced by a call to such a procedure depends functionally on just the meaning (i.e., input/output behavior) of each procedural parameter together with the value of each data parameter.[1] A procedure is said to be *extensional* in a particular (syntactic) argument if and only if the procedure uses this argument only for its procedural meaning. Here are two examples. Let N denote the natural numbers, let $\langle \varphi_p \rangle_{p \in \mathsf{N}}$ be an acceptable programming system (see Section 2) of the partial recursive functions over N, let $apply = \lambda p, x \in \mathsf{N}.\varphi_p(x)$ (i.e., $apply(p, x)$ simply returns the result of running program p on input x), and, finally, let $g = \lambda p, x \in \mathsf{N}.(\varphi_p(x) + p)$. Clearly, $apply$ is extensional in its first argument, but g is not.[2] We call this view of higher-order procedures the *glass-box approach*.

Most programming language texts implore you *not* to think of procedural parameters as syntactic objects. Their view is that a procedural parameter is considered as being contained in a *black box* that one can query for its input/output behavior, but one cannot see the code inside. Moreover, ML, Scheme, Haskell, etc., work to enforce this black-box view of procedures. A key rationale for this is that it is *very* difficult to tell whether a procedure over syntactic objects is extensional in a certain argument. Putting the procedural parameters into black boxes, on the other hand, guarantees that procedural parameters are used *only* for their input/output behavior.

There is an important question as to whether this black-box approach limits one's computing power. By looking at the syntax of a procedure, one might conceivably learn more about its input/output behavior than by simply querying a black box containing the procedure. So, does every glass-box style higher-order procedure correspond to a black-box style higher-order procedure? For type-2 there are two beautiful affirmative answers from recursion theory, given as Theorems 2 and 3 below. Before stating these theorems, we introduce some terminology and conventions.

Terminology: $A \rightharpoonup B$ (respectively, $A \to B$) denotes the collection of partial (respectively, total) functions from A to B. $\mathcal{PR}$ (respectively, $\mathcal{R}$) denotes the collection of partial (respectively, total) recursive functions over N. Let $\langle \varphi_p \rangle_{p \in \mathsf{N}}$ be as before. For a partial function α of any type, the notation: $\alpha(\textit{arguments})\downarrow$ (respectively, $\alpha(\textit{arguments})\uparrow$) means that α is defined (respectively, undefined) for the given arguments.

General Convention: In this paper we restrict our attention to functionals that take one func-

[1] In this paper we will not worry about the possibility of the value itself being a procedure.

[2] Where the "meaning" of p is taken to be φ_p.

tion argument and one numeric argument and return a numeric result. Generalizing the definitions and results below to functionals that take more than one function argument and multiple numeric arguments is straightforward, but adds nothing except notation to the discussion here.

Definition 1.

(a) Suppose $\mathcal{C} \subseteq \mathcal{PR}$. $\Gamma: \mathcal{C} \times \mathsf{N} \rightharpoonup \mathsf{N}$ is an *effective operation* on $\mathcal{C}$ if and only if, for some partial recursive $\alpha: \mathsf{N} \rightharpoonup \mathsf{N}$, we have that, for all p with $\varphi_p \in \mathcal{C}$ and all a, $\Gamma(\varphi_p, a) = \alpha(p, a)$; α is said to *determine* Γ.[3] When $\mathcal{C} = \mathcal{PR}$, Γ is called, simply, an *effective operation*. Γ is a *total effective operation on* $\mathcal{C}$ if and only if Γ is an effective operation on $\mathcal{C}$ with the additional property that, for all $\psi \in \mathcal{C}$ and all $a \in \mathsf{N}$, $\Gamma(\psi, a)$ is defined. *Convention:* Whenever we speak of a *computation of an effective operation* we shall mean the computation of a particular, fixed program for a partial recursive function that determines the effective operation.

(b) $\Gamma: (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{N} \rightharpoonup \mathsf{N}$ is a *partial recursive functional* if and only if there is an oracle Turing machine $\mathbf{M}$ (with a function oracle) such that, for all $\alpha: \mathsf{N} \rightharpoonup \mathsf{N}$ and all $a \in \mathsf{N}$, $\Gamma(\alpha, a) = \mathbf{M}(\alpha, a)$.[4]

(c) Let $\langle \sigma_i \rangle_{i \in \mathsf{N}}$ be a canonical indexing of finite functions. $\Gamma: (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{N} \rightharpoonup \mathsf{N}$ is an *effective continuous functional* if and only if there is an r.e. set A such that, for all $\alpha: \mathsf{N} \rightharpoonup \mathsf{N}$ and all $a, z \in \mathsf{N}$, $\Gamma(\alpha, a)\downarrow = z \iff (\exists i)[\, \sigma_i \subseteq \alpha \text{ and } \langle i, a, z \rangle \in A\,]$.[5]

(d) Two functionals Γ and Γ' *correspond* on $\mathcal{C}$ if and only if, for all $\psi \in \mathcal{C}$ and all a, $\Gamma(\psi, a) = \Gamma'(\psi, a)$.

(e) Two classes of functionals $\mathbf{F}_0$ and $\mathbf{F}_1$ are said to *correspond on* $\mathcal{C}$ if and only if each Γ_0 in $\mathbf{F}_0$ corresponds on $\mathcal{C}$ to some $\Gamma_1 \in \mathbf{F}_1$ and each Γ_1 in $\mathbf{F}_1$ corresponds on $\mathcal{C}$ to some $\Gamma_0 \in \mathbf{F}_0$. $\diamond$

Theorem 2 (The Myhill-Shepherdson Theorem [MS55]). *The following classes of functionals all correspond on* $\mathcal{PR}$*:*

(a) The effective operations on $\mathcal{PR}$*.*

(b) The effective continuous functionals.

[3] Note that α is extensional in its first argument with respect to p with $\varphi_p \in \mathcal{C}$.

[4] If, in the course of its computation, such an $\mathbf{M}$ queries its oracle α on an x for which α is undefined, then at that point, $\mathbf{M}$ goes undefined.

[5] Rogers [Rog67] calls these *recursive functionals*. Intuitively, a computation for an effective continuous functional can concurrently query its function argument α about multiple values and, when the functional has enough information about α, it can produce an answer without having to wait for all of the answers to its queries to come in. A computation for a partial recursive functional, in contrast, is constrained to make *sequential* queries to its function argument, that is, the answer to one query must be received before another can be issued. Effective continuous functionals are strictly more general than partial recursive functionals. For example,

$$\mathbf{OR}_{\|} = \lambda\alpha, x.\begin{cases} 1, & \text{if } \alpha(x)\downarrow \neq 0 \text{ or } \alpha(x+1)\downarrow \neq 0; \\ 0, & \text{if } \alpha(x)\downarrow = \alpha(x+1)\downarrow = 0; \\ \uparrow, & \text{otherwise;} \end{cases} \tag{1}$$

is an effective continuous, but not a partial recursive, functional. See Rogers's [Rog67] or Odifreddi's [Odi89] texts for detailed discussions of these notions.

Theorem 3 (The Kreisel-Lacombe-Shoenfield Theorem [KLS57]). *The following classes of functionals all correspond on $\mathcal{R}$:*

(a) The total effective operations on $\mathcal{R}$.

(b) The effective continuous functionals that are total on $\mathcal{R}$.

(c) The partial recursive functionals that are total on $\mathcal{R}$.[6]

Theorem 2 is part of the foundations of programming language semantics (see, for example, [Sco75, pp. 190–193]). The two theorems say that the power of effective operations (a syntactic/glass-box notion) is no greater than the power effective continuous functionals (a semantic/black-box notion) in two settings considered in the theorems. Thus, treating procedural parameters as black boxes does not lose one computing power.

But what of efficiency? It is quite conceivable that in computing an effective operation, where one has access to the syntax of the argument procedures, one could gain an efficiency advantage over any corresponding black-box-style, higher-order procedure. Theorems 2 and 3 and their standard proofs are uninformative on this point. This paper makes a start at examining this question by considering whether analogues of Theorem 3 hold for the Mehlhorn-Cook class of type-2 feasible functionals, see Section 3 below. After establishing some general conventions in Section 2, Section 3 discusses the basic polynomial-time functionals, and Sections 4 and 5 then describe two different approaches to addressing this question. Section 6 provides proofs of the results of Sections 4 and 5. Finally, Section 7 indicates some other paths for exploration.

Acknowledgments Thanks to Jin-Yi Cai, John Case, Robert Irwin, Bruce Kapron, Stuart Kurtz, Ken Regan, Alan Selman, and Anil Seth for discussing this work at various stages of its development. Thanks also to the anonymous referees for suggesting some important improvements and pointing out one major error. Special thanks to Neil Jones and the TOPPS group at DIKU for letting me spend a week at DIKU to discuss my ideas and to Peter O'Hearn for many, many discussions on these and related topics. The research for this paper was partially supported by National Science Foundation grant CCR-9522987.

2. Notation, Conventions, and Such

Basics We identify each $x \in \mathsf{N}$ with its dyadic representation over $\{\,\mathbf{0}, \mathbf{1}\,\}$ and define $|x|$ to be the length of its representation. We let $\langle \cdot, \cdot \rangle$ denote a standard, polynomial-time pairing function with the property that, for all x and $y \geq 1$, $\max(x, y) < \langle x, y \rangle$. (Rogers' [Rog67] pairing function will do.) Following Kapron and Cook [KC91, KC96] we define the length of a function $\alpha\colon \mathsf{N} \rightharpoonup \mathsf{N}$ (denoted $|\alpha|$) to be the function $\lambda n\,.\,\max\{\,|\alpha(x)| : |x| \leq n\,\}$. (Note that if α is partial, then $|\alpha|$ is almost everywhere undefined.)

Machines M (with and without decorations) varies over multi-tape, deterministic Turing machines (TMs). $\mathbf{M}$ and M (with and without decorations) vary over oracle, multi-tape, determin-

[6] The totality assumptions are necessary. Friedberg [Fri58, Rog67] has an example of a (nontotal) effective operation on $\mathcal{R}$ that fails to correspond to any effective continuous functional.

istic Turing machines (OTMs) where the oracles are partial functions. We shall play a bit loose with the TM and OTM models: we will speak of Turing machines as having subroutines, counters, arrays, etc. All of these can be realized in the standard model in straightforward ways with polynomial overhead, which suffices for the purposes of this paper. In our one bit of fussiness, we follow Kapron and Cook's [KC91, KC96] conventions for assigning costs to OTM operations. Under these conventions, the cost of an oracle query $\alpha(x)$=? is $\max(1, |\alpha(x)|)$, if $\alpha(x)\downarrow$, and ∞, if $\alpha(x)\uparrow$. Every other OTM operation has unit cost.

The Standard Acceptable Programming System and Complexity Measures We assume that our standard acceptable programming system, $\langle\varphi_p\rangle_{p\in\mathsf{N}}$, is based on an indexing of Turing machines. (For more on acceptable programming systems, see Rogers [Rog67]—where they are called acceptable numberings, or Machtey and Young [MY78], or Royer and Case [RC94].) $\langle\Phi_p\rangle_{p\in\mathsf{N}}$ denotes the standard run-time measure on Turing machines, our standard complexity measure associated with $\langle\varphi_p\rangle_{p\in\mathsf{N}}$. Thus, for all p and x, $\Phi_p(x)$ denotes the run time of (Turing machine) program p on input x and, hence, $\Phi_p(x) \geq \max(|x|, |\varphi_p(x)|)$ where the max is ∞ when $\varphi_p(x)\uparrow$. For each p and x, define

$$\Phi^*_p(x) \quad = \quad \max(|p|, \Phi_p(x)).$$

Intuitively, under Φ^* the costs of loading the program, reading the input, and writing the output are all part of the cost of running program p on input x. Also, for each p and n, define:

$$\overline{\Phi}_p(n) \quad = \quad \max\{\, \Phi_p(x) : |x| \leq n \,\}.$$

$$\overline{\Phi}^*_p(n) \quad = \quad \max\{\, \Phi^*_p(x) : |x| \leq n \,\}.$$

3. Basic Polynomial-Time Functionals

Mehlhorn [Meh76] introduced a class of type-2 functionals to generalize the notion of Cook reducibility from reductions between sets to reductions between functions. His definition was based on a careful relativization of Cobham's [Cob65] syntactic definition of polynomial-time. Some years later Cook and Urquhart [CU89, CU93], in extending work of Buss [Bus86], defined an equivalent class of type-2 functionals, as well as analogous functionals of type 3 and above. This class of functionals, which they called the (type-2) *basic feasible functionals*, was developed by Cook and co-workers in a series of papers; see, for example, [Coo91, CK90, KC91, KC96]. Kapron and Cook's 1991 paper [KC91, KC96] is of particular importance here, as that paper introduced the first natural machine characterization of the type-2 basic feasible functionals, stated as Theorem 6 below.

Definition 4 (Kapron and Cook [KC91, KC96]). A *second-order polynomial* over type-1 variables $f_0, \ldots, f_m$ and type-0 variables $x_0, \ldots, x_n$ is an expression of one of the following five forms:

1. a 2. x_i 3. $\mathbf{q}_1 + \mathbf{q}_2$ 4. $\mathbf{q}_1 \cdot \mathbf{q}_2$ 5. $f_j(\mathbf{q}_1)$

where $a \in \mathsf{N}$, $i \leq n$, $j \leq m$, and $\mathbf{q}_1$ and $\mathbf{q}_2$ are second-order polynomials over $\vec{f}$ and $\vec{x}$. The value of a second-order polynomial as above on $g_0, \ldots, g_m \colon \mathsf{N} \to \mathsf{N}$ and $a_0, \ldots, a_n \in \mathsf{N}$ is the obvious thing. $\diamond$

Convention: In the following second-order polynomials will be over one type-1 variable and one type-0 variable unless we explicitly assert otherwise.

Definition 5 (Kapron and Cook [KC91, KC96]). $\Gamma \colon (\mathsf{N} \to \mathsf{N}) \times \mathsf{N} \to \mathsf{N}$ is a *basic polynomial-time functional* if and only if there is an OTM $\mathbf{M}$ and a second-order polynomial $\mathbf{q}$ such that, for all $g \colon \mathsf{N} \to \mathsf{N}$ and $a \in \mathsf{N}$:

1. $\Gamma(g, a) = \mathbf{M}(g, a)$.
2. On input (g, a), $\mathbf{M}$ runs within $\mathbf{q}(|g|, |a|)$ time. $\diamond$

Theorem 6 (Kapron and Cook [KC91, KC96]). *The class of type-2 basic feasible functionals corresponds on* $\mathsf{N} \to \mathsf{N}$ *to the class of basic polynomial-time functionals.*

This is a lovely and important result. However, there are two difficulties with Definition 5. First, the bound $\mathbf{q}(|\alpha|, |a|)$ is nonsensical when α is not total. Second, even for a total function g, the bound $\mathbf{q}(|g|, |a|)$ seems problematic. This is because $|g|(n)$ is defined by a max over $2^{n+1} - 1$ many values. Hence, as Kapron and Cook point out, the bound $\mathbf{q}(|g|, |a|)$ may not be feasibly computable even when g is polynomial-time.[7] Seth [Set92, Set94] resolved both of these problems by formalizing the clocking notion (Definition 7) and proving the characterization (Theorem 8) below. (As Seth notes, these ideas are implicit in [KC91, KC96].) The idea behind the clocking scheme is that in running a machine $\mathbf{M}$ clocked by a second-order polynomial $\mathbf{q}$, one keeps a running lower approximation to $\mathbf{q}(|g|, |a|)$ based on the information on g gathered from $\mathbf{M}$'s queries during the computation. Under the clocking scheme, $\mathbf{M}$'s computation runs to completion if and only if, for each step of $\mathbf{M}$, $\mathbf{M}$'s run time up to this step is less than the current approximation to $\mathbf{q}(|g|, |a|)$. (For a discussion of clocked programming systems for first-order complexity theory, see Chapter 4 of Royer and Case's monograph [RC94].)

Notation: Suppose q is an ordinary polynomial over two variables. For each $d \in \mathsf{N}$, define $q^{[d]}$ to be the second-order polynomial over f and x as follows:

$$\begin{aligned} q^{[0]}(f, x) &= q(0, x). \\ q^{[d+1]}(f, x) &= q(f(q^{[d]}(f, x)), x). \end{aligned}$$

[7] The functional $\lambda g, \mathbf{0}^n \,.\, |g|(n)$ fails to be basic feasible, although the g's that standard diagonal constructions produce to witness this failure are not polynomial-time computable. Lance Fortnow pointed out that if $\mathrm{DTIME}(2^{\mathrm{poly}}) \neq \mathrm{NTIME}(2^{\mathrm{poly}})$, then there is a g that is polynomial-time computable but such that $\lambda \mathbf{0}^n \,.\, |g|(n)$ is not. *Proof Sketch:* Suppose $\mathrm{DTIME}(2^{\mathrm{poly}}) \neq \mathrm{NTIME}(2^{\mathrm{poly}})$. Then there is a $T \subseteq \mathbf{0}^*$ (a *tally language*) that is in (NP$-$P) [Boo74]. Suppose that Q is a polynomial-time decidable predicate and $c \in \mathsf{N}$ are such that $T = \{\, \mathbf{0}^n : (\exists y : |y| = c \cdot n^c)\, Q(\mathbf{0}^n, y) \,\}$. Define, for all y,

$$g(y) = \begin{cases} 2^{n+1}, & \text{if, for some } n,\ |y| = c \cdot n^c \text{ and } Q(\mathbf{0}^n, y); \\ 0, & \text{otherwise.} \end{cases}$$

Clearly, g is polynomial-time computable. Observe that, for each n, $\mathbf{0}^n \in T$ if and only if $|g|(\mathbf{0}^{c \cdot n^c}) = n + 1$. Since $T \notin \mathrm{P}$, $\lambda \mathbf{0}^n \,.\, |g|(n)$ cannot be polynomial-time computable.

A straightforward argument shows that, for each second-order polynomial $\mathbf{q}$ over f and x, there is a polynomial q and a $d \in \mathsf{N}$ such that, for all g and a, $\mathbf{q}(|g|, |a|) \leq q^{[d]}(|g|, |a|)$.

Definition 7.

(a) Suppose $\mathbf{M}$ is an OTM, q is an ordinary polynomial over two variables, and $d \in \mathsf{N}$. Let $\mathbf{M}_{q,d}$ be the OTM that, on input (α, a), operates as follows. $\mathbf{M}_{q,d}$ maintains a counter, clock, and two arrays $\mathsf{x}[0..d]$ and $\mathsf{y}[0..d-1]$. $\mathbf{M}_{q,d}$ maintains the following invariants, where $i = 0, \ldots, d-1$.

$$\begin{aligned} \mathsf{x}[0] &= q(0, |a|). \\ \mathsf{x}[i+1] &= q(\mathsf{y}[i], |a|). \\ \mathsf{y}[i] &= \max\left(\left\{ |\alpha(x)| : \begin{array}{l} |x| \leq \mathsf{x}[i] \text{ and the query} \\ \alpha(x) =? \text{ has been made} \end{array} \right\}\right). \end{aligned}$$

(For each $i \leq d$, $\mathsf{x}[i]$ is our lower approximation to $q^{[i]}(|\alpha|, |a|)$ and, for each $i < m$, $\mathsf{y}[i]$ is our lower approximation to $|\alpha|(q^{[i]}(|\alpha|, |a|))$.) On start up, $\mathbf{M}_{q,d}$ sets clock and each $\mathsf{y}[i]$ to 0 and sets each $\mathsf{x}[i]$ to $q(0, |a|)$. Then, $\mathbf{M}_{q,d}$ simulates $\mathbf{M}$ step-by-step on input (α, a). For each step of $\mathbf{M}$ simulated:

- If the step of $\mathbf{M}$ just simulated is the query $\alpha(x)$=?, then:
  - if $\alpha(x)\uparrow$, then $\mathbf{M}_{q,d}(\alpha, a)$ is undefined, and
  - if $\alpha(x)\downarrow$, then, if necessary, $\mathbf{M}_{q,d}$ recomputes the $\mathsf{x}[i]$'s and $\mathsf{y}[i]$'s to re-establish the invariants.
- If, in the step of $\mathbf{M}$ just simulated, $\mathbf{M}$ halts with output y, then $\mathbf{M}_{q,d}$ outputs y and halts.
- If $\mathbf{M}$ did not halt in the step just simulated, then the cost of the step is added to clock and, if $\mathsf{clock} < \mathsf{x}[d]$, the simulation continues; otherwise, $\mathbf{M}_{q,d}$ outputs 0 and halts.

(b) We say that $\Gamma\colon (\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{N} \rightharpoonup \mathsf{N}$ is a *clocked basic polynomial-time functional* if and only if Γ is computed by one of the $\mathbf{M}_{q,d}$'s as defined above. $\diamond$

Note that a clocked basic polynomial-time functional has domain $(\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{N}$, whereas a basic polynomial-time functional has just $(\mathsf{N} \to \mathsf{N}) \times \mathsf{N}$ as its domain. Seth's notion is very conservative and constructive as compared to Definition 5 which on the surface seems to allow for machines that nonconstructively obey their time bounds. The following characterization is thus a little surprising and very pleasing.

Theorem 8 (Seth [Set92]). *The class of clocked basic polynomial-time functionals correspond on* $(\mathsf{N} \to \mathsf{N})$ *to the class of basic polynomial-time functionals.*

Therefore, the classes of type-2 basic feasible functionals, basic polynomial-time functionals, and clocked basic polynomial-time functionals all correspond on $\mathsf{N} \to \mathsf{N}$, which is good evidence that this class of functionals is robust.

4. Polynomial-Time Operations

4.1. Definitions

We now consider how to define a sensible polynomial-time analog of an effective operation. We begin by reconsidering $apply: \mathsf{N}^2 \rightharpoonup \mathsf{N}$ from Section 1. For most straightforward implementations, the cost of computing $apply(p, x)$ is at least $\Phi_p(x)$ and is bounded above by $(\Phi_p(x))^{O(1)}$. It thus seems reasonable that an account of the cost of computing an effective operation would include some dependence on the costs of running the program argument on various values during the course of the computation. The next proposition shows that this dependence is in fact necessary to obtain a nontrivial notion.

Proposition 9. *Suppose i is such that φ_i determines a total effective operation on $\mathcal{R}$ and that there is a second-order polynomial* **q** *such that, for all p with φ_p total and all x, $\Phi_i(p, x) \leq \mathbf{q}(|\varphi_p|, \max(|p|, |x|))$. Then, there is a polynomial-time computable $f: \mathsf{N} \to \mathsf{N}$ such that, for all p with φ_p total and all x, $\varphi_i(p, x) = f(x)$.*

We thus introduce the following definition which is modeled after Definition 5. Recall that, for all p, x, and n, $\Phi^*_p(x) \geq \max(|p|, |x|, |\varphi_p(x)|)$ and $\overline{\Phi}^*_p(n) = \max\{\, \Phi^*_p(x) : |x| \leq n \,\}$.

Definition 10. $\Gamma: \mathcal{R} \times \mathsf{N} \to \mathsf{N}$ is a *polynomial-time operation* if and only if there exist a partial recursive $\alpha: \mathsf{N}^2 \rightharpoonup \mathsf{N}$ and a second-order polynomial **q** such that, for all p with φ_p total and all $a \in \mathsf{N}$:

1. $\Gamma(\varphi_p, a) = \alpha(p, a)$.
2. $\alpha(p, a)$ is computable within time $\mathbf{q}(\overline{\Phi}^*_p, |a|)$.[8] ◇

Clearly, each basic polynomial-time functional corresponds on $\mathcal{R}$ to some polynomial-time operation. However, Definition 10 is not terribly satisfactory. It has the same problems noted of Definition 5—only worse, as it will turn out. To address these problems we introduce a clocked version of polynomial-time operation analogous to Seth's notion of Definition 7. But there is a difficulty in the way of this. In the computation of an effective operation, there are neither oracle calls nor reliable ways of telling when, for particular p_0 and x_0, $\varphi_{p_0}(x_0)$ is evaluated. Hence, it is a puzzle how a clocking mechanism is to gather appropriate information to approximate $\mathbf{q}(\overline{\Phi}^*_p, |a|)$. Our solution is an appeal to bureaucracy—we make clocked Turing machines that compute effective operations fill out standardized forms to justify their expenses. That is, we equip the machines computing effective operations with UNIV, a standard subroutine that computes $\lambda p, x.\varphi_p(x)$. When UNIV is called on arguments (p, x_0) (where p is the machine's procedural parameter), we use the number of steps UNIV simulates of φ-program p on input x_0

[8] Note that this notion (and the notions of Definitions 11(d), 16(c), and 21(d)) is implicitly parameterized by our choices of φ and Φ. Also note that by rights these functionals should be called the '*basic* polynomial-time operations' to indicate a bit of reservation about this class being the "correct" polynomial-time analogs of effective operations. While this reservation is quite reasonable, the terminology is already too long-winded. Thus, I have avoided using "basic" in this and the other terminology of Sections 4 and 5.

as data for our lower approximation of $\mathbf{q}(\overline{\Phi}^*_p, |a|)$. Thus, one of these clocked machines has, at each point of each computation, an observable, verifiable justification for the amount of time it has consumed. These machines for computing effective operations are perfectly free to use means other than UNIV to evaluate $\varphi_p(x_0)$ for various x_0, but UNIV is the *only* means for justifying big run times to the clocking mechanism. Here are the details.

Definition 11.

(a) Let UNIV denote a fixed TM subroutine that takes two arguments, p_0 and x_0, and step-by-step simulates φ-program p_0 on input x_0 until, if ever, the simulation runs to its conclusion, at which time UNIV writes $\varphi_{p_0}(x_0)$ and $\Phi_{p_0}(x_0)$ on two separate tapes and erases all of its other tapes. We assume UNIV on arguments (p_0, x_0) runs in $(\Phi^*_{p_0}(x_0))^{O(1)}$ time.

(b) A *special Turing machine* (STM) M is a Turing machine defined as follows. M takes two inputs (p, x). M includes UNIV as a subroutine. M's instructions outside of UNIV do not write on any of UNIV's tapes except UNIV's input tape. When M is running UNIV on arguments (p_0, x_0) and $p_0 = p$, we say that M is making a *normal query*.[9]

(c) Suppose M is an STM, q is a polynomial over two variables, and $d \in \mathsf{N}$. Let $M_{q,d}$ be the STM that, on input (p, a), operates as follows. $M_{q,d}$ maintains a counter clock and two arrays x[0..d] and y[0..$d-1$]. $M_{q,d}$ maintains the following invariants, where $i = 0, \ldots, d-1$:

$$\begin{aligned} \mathsf{x}[0] &= q(0, |a|). \\ \mathsf{x}[i+1] &= q(\mathsf{y}[i], |a|). \\ \mathsf{y}[i] &= \max\left(\left\{ \Phi^*_p(x_0) \ : \ \begin{array}{l} |x_0| \le \mathsf{x}[i] \text{ and the normal query} \\ \varphi_p(x_0) = ? \text{ has been made} \end{array} \right\}\right). \end{aligned}$$

On start up, $M_{q,d}$ initializes clock, x, and y exactly as $\mathbf{M}_{q,d}$ does. Then, $M_{q,d}$ simulates M step-by-step on input (p, a). For each step of M simulated:

1. If the step of M just simulated was the last step of a normal query (i.e., we are returning from a call to UNIV on (p, x_0) for some x_0), then, if necessary, $M_{q,d}$ recomputes the x[i]'s and y[i]'s to re-establish the invariants.

2. If, in the step of M just simulated, M halts with output y, then $M_{q,d}$ outputs y and halts.

3. If M did not halt in the step just simulated, then the value of clock is increased by 1 and, if clock $<$ x[d] *or if the step of M just simulated was part of a normal query*, then the simulation continues; otherwise, $M_{q,d}$ outputs 0 and halts.

(d) Γ: $(\mathsf{N} \rightharpoonup \mathsf{N}) \times \mathsf{N} \rightharpoonup \mathsf{N}$ is a *clocked polynomial-time operation* if and only if there exists an $M_{q,d}$ such that, for all p and x, $\Gamma(\varphi_p, x) = M_{q,d}(p, x)$. $\diamond$

[9] Thus, a normal query to UNIV is like a query to an oracle for the φ-universal function to find the value of $\varphi_p(x_0)$, except we are bereft of divine inspiration and have to work out the answer to the query.

Note that a clocked polynomial-time operation has domain $\mathcal{PR} \times \mathsf{N}$, whereas a polynomial-time operation has just $\mathcal{R} \times \mathsf{N}$ as its domain. However, a clocked polynomial-time operation when restricted to $\mathcal{R} \times \mathsf{N}$ corresponds to some polynomial-time operation.[10]

4.2. Comparisons

Section 3 concluded by stating the correspondence on $\mathsf{N} \to \mathsf{N}$ of the classes type-2 basic feasible functionals, basic polynomial-time functionals, and clocked basic polynomial-time functionals. Here, complexity theoretic conundrums preclude having such a neat or conclusive story.

We first show that if $\mathrm{P} = \mathrm{NP}$, then the type-2 basic feasible functionals and the clocked polynomial-time operations *fail* to correspond on $\mathcal{R}$. We make use of the following functional. For each $\alpha\colon \mathsf{N} \rightharpoonup \mathsf{N}$ and $x \in \mathsf{N}$, define:

$$\Gamma_0(\alpha, x) \quad = \quad \begin{cases} \uparrow, & \text{if (i): for some } y \in \{\mathbf{0}, \mathbf{1}\}^{|x|},\ \alpha(y)\uparrow; \\ 1, & \text{if (ii): } \textit{not} \text{ (i) and } (\exists y \in \{\mathbf{0}, \mathbf{1}\}^{|x|})[\ \alpha(y) \text{ is odd }]; \\ 0, & \text{if (iii): } \textit{not} \text{ (i) and } (\forall y \in \{\mathbf{0}, \mathbf{1}\}^{|x|})[\ \alpha(y) \text{ is even }]. \end{cases}$$

Proposition 12.

(a) The restriction of Γ_0 *to* $(\mathsf{N} \to \mathsf{N}) \times \mathsf{N}$ *is not basic feasible.*

(b) If $\mathrm{P} = \mathrm{NP}$*, then* Γ_0 *is a clocked polynomial-time operation.*

Thus, if the analog of the Kreisel-Lacombe-Shoenfield Theorem holds for the classes of functionals here under consideration, then $\mathrm{P} \neq \mathrm{NP}$. Remark 20 in Section 6 notes that one can weaken the $\mathrm{P} = \mathrm{NP}$ hypothesis of Proposition 12(b).

Similar difficulties arise in comparing the clocked and unclocked polynomial-time operations.

Proposition 13. *If* $\mathrm{P} = \mathrm{NP}$*, then the polynomial-time operations correspond on* $\mathcal{R}$ *to the clocked polynomial-time operations.*

Remark 14. The version of this paper that appeared in the *10th Annual IEEE Structure in Complexity Theory Conference* proceedings asserted: (i) there is an oracle relative to which the polynomial-time operations correspond on $\mathcal{R}$ to the basic feasible functionals and (ii) there is an oracle relative to which the polynomial-time operations fail to correspond on $\mathcal{R}$ to the clocked polynomial-time operations. The arguments for both (i) and (ii) turned out to *wrong*. As far as the author can tell, the standard techniques for oracle construction seem to run into trouble on attempts to prove (i) and (ii).[11] Based on general pessimism, we conjecture that both (i) and (ii) are true. ◇

[10] Also note that this clocking scheme is based on *sequential* queries to UNIV. This causes a problem for nontotal function arguments. For example, the functional $\mathbf{OR}_{\|}$ from (1) is intuitively feasibly computable and is a polynomial-time operation, but it is easy to show that $\mathbf{OR}_{\|}$ is *not* a clocked polynomial-time operation.

[11] The difficulty is that in order to satisfy the positive requirements in such a construction, one is forced to code information about computations of *all* the programs of some relativized acceptable programming system. This information is too dense and too deep to allow any room in the oracle to satisfy the construction's negative requirements.

The question with which we started was whether, in a polynomial-time setting, effective operations have an efficiency advantage over black-box style functionals. The above results demonstrate that there is little hope of resolving this question with present-day complexity theory. Since we have hit an apparent dead end with the original question, let us change the question a bit and ask instead to what extent can one open up the black boxes and still obtain a *provable* equivalence with the black-box models. The next section investigates one approach to this.

5. Functionals Determined by Computations over Computations

5.1. Definitions

Machines computing black-box style functionals have access only to the input/output behavior of their procedural parameters. Here we consider a style of functional where the machines computing them have access only to the *computational* behavior of their procedural parameter. Ideally, what we would like is a model where a machine computing a functional has the text of its procedural parameter hidden, but in which the machine can run its procedural parameter step-by-step on various arguments and observe the results, i.e., observe traces of computations evolve. In this paper we settle for a simplified/sanitized version of the above model that is still in the same spirit. In the model we use, machines computing a functional are supplied with an oracle that corresponds to the functional's procedural parameter as follows. When queried on $(x, \mathbf{0}^k)$, the oracle returns the result of running the procedural parameter on argument x, *provided* the procedural parameter produces an answer within k steps; if this is not the case, then the oracle returns $\star$, indicating "no answer yet." (Think of this model as providing black boxes with cranks attached that you have to turn a requisite number of times to receive an answer.) Below we formalize *shreds* ($\approx$ faint traces), a class of functions that corresponds to such oracles, and *computation systems*, the recursion/complexity theoretic inspiration of shreds.

Notation: Define $\mathsf{N}_\star = \mathsf{N} \cup \{\star\}$, where $\star \notin \mathsf{N}$. Let ω denote a copy of N where the elements of ω are understood to be represented in unary over $\mathbf{0}^*$.

Definition 15.

(a) Suppose $\widehat{\varphi}$ is an acceptable programming system and $\widehat{\Phi}$ is a complexity measure associated with $\widehat{\varphi}$. The *computation system* for $\widehat{\varphi}$ and $\widehat{\Phi}$ is the recursive function $\widehat{\chi}\colon \mathsf{N}^2 \times \omega \to \mathsf{N}_\star$ defined by

$$\widehat{\chi} \;=\; \lambda p, x, \mathbf{0}^k .\, \begin{cases} \widehat{\varphi}_p(x), & \text{if } \widehat{\Phi}_p(x) \leq k; \\ \star, & \text{otherwise.} \end{cases} \tag{2}$$

We usually write $\widehat{\chi}_p(x, \mathbf{0}^k)$ for $\widehat{\chi}(p, x, \mathbf{0}^k)$. Let χ be the computation system associated with φ and Φ, our standard, Turing machine-based acceptable programming system and associated complexity measure.

One might try to get around this difficulty by a slight complexity-theoretic subversion of Rice's Theorem in the relativized world. But Rice's Theorem is such a simple and strong property of acceptable programming systems, it seems to defy any such subversion.

(b) A function $s\colon \mathsf{N} \times \omega \to \mathsf{N}_\star$ is a *shred* if and only, if for each x, either

(i) for all $\mathbf{0}^k$, $s(x, \mathbf{0}^k) = \star$, or else

(ii) there are y and k_0 such that, for all $k < k_0$, $s(x, \mathbf{0}^k) = \star$ and, for all $k \geq k_0$, $s(x, \mathbf{0}^k) = y$.

(Thus, each χ_p is a shred.)

(c) Suppose s is a shred. We define:

$$\begin{aligned} \kappa s &= \lambda x\,.\,(\mu \mathbf{0}^k)[s(x, \mathbf{0}^k) \neq \star]. \\ \iota s &= \lambda x\,.\,s(x, \kappa s(x)). \\ \overline{\kappa s} &= \lambda n\,.\,\max\{\, \kappa s(x) : |x| \leq n \,\}. \end{aligned}$$

It is understood that if $\kappa s(x)\!\uparrow$, then $\iota s(x)\!\uparrow$ too. (Thus, for each p, $\iota \chi_p = \varphi_p$, $\kappa \chi_p = \Phi_p$, and $\overline{\kappa \chi_p} = \overline{\Phi}_p$.)

(d) $\mathcal{S}_{\text{all}}$ denotes the collection of all shreds.

(e) For each $\mathcal{S} \subseteq \mathcal{S}_{\text{all}}$, $\iota\mathcal{S}$ denotes $\{\, \iota s : s \in \mathcal{S} \,\}$ and $\text{tot}(\mathcal{S})$ denotes $\{\, s \in \mathcal{S} : \iota s \text{ is total} \,\}$.

(f) For each computation system $\widehat{\chi}$, $\mathcal{S}_{\widehat{\chi}}$ denotes $\{\, \widehat{\chi}_p : p \in \mathsf{N} \,\}$.

(g) $\mathcal{S}_{\text{TM}}$ denotes $\mathcal{S}_\chi$, i.e., the collection of shreds based on Turing machine computations. $\diamond$

The right-hand side of (2) is a familiar tool from numerous recursion and complexity theoretic arguments. In most of these arguments the right-hand side of (2) embodies all the information needed about the computations of $\widehat{\varphi}$-programs. Hence, for such arguments, shreds represent an adequate abstraction of computations.

Our next goal is to formalize an analog of the notion of effective operation where shreds take the role played by programs in Definition 1(a).

Notation: $\mathcal{S}$ will range over subsets of $\mathcal{S}_{\text{all}}$. M will range over OTMs whose function oracles range over $\mathcal{S}_{\text{all}}$.

Definition 16. Suppose $\mathcal{S}$ is such that $\mathcal{R} \subseteq \iota\mathcal{S}$ and suppose $\Gamma\colon \mathcal{R} \times \mathsf{N} \rightharpoonup \mathsf{N}$.

(a) We say that an OTM M is *extensional* with respect to $\mathcal{S}$ if and only if, for all s and $s' \in \mathcal{S}$ and all $a \in \mathsf{N}$, if $\iota s = \iota s'$, then $\mathsf{M}(s, a) = \mathsf{M}(s', a)$.

(b) We say that Γ is a *recursive shred-operation* with respect to $\mathcal{S}$ if and only if there is an OTM M that is extensional with respect to $\mathcal{S}$ such that, for all $s \in \mathcal{S}$ with $\iota s \in \mathcal{R}$ and all $a \in \mathsf{N}$, $\Gamma(\iota s, a) = \mathsf{M}(s, a)$; we say that M *determines* Γ.

(c) Γ is a *polynomial-time shred-operation* with respect to $\mathcal{S}$ if and only if there is an OTM M and a second order polynomial $\mathbf{q}$ such that, for all $s \in \mathcal{S}$ with $\iota s \in \mathcal{R}$ and all $a \in \mathsf{N}$:

1. $\Gamma(\iota s, a) = \mathsf{M}(s, a)$.

2. On input (s, a), M runs within $\mathbf{q}(\overline{\kappa s}, |a|)$ time. $\diamond$

For each of the notions just defined, when the collection $\mathcal{S}$ is understood we usually suppress mention of it.[12]

Definition 16(c) suffers from difficulties analogous to the problems with Definitions 5 and 10—the bound $\mathbf{q}(\overline{\kappa s}, |a|)$ may not be feasibly computable and the totality restriction is a nuisance. So, as in Sections 3 and 4, here we introduce a clocked version of the primary functional notion. Our clocking scheme is again based on the petty bureaucratic measure of having clocked machines fill out standardized forms to justify their expenses. In the present case this means that we equip OTMs computing our clocked functionals with a subroutine RUN, which is as follows. Suppose $s \in \mathcal{S}_{\text{all}}$ is the function oracle of one of these OTM's. When an OTM calls RUN on $x \in \mathsf{N}$, the result is either

1. $\langle s(x, \mathbf{0}^{2^k}), \mathbf{0}^{2^k}\rangle$ is returned, if there exists a k' such that $s(x, \mathbf{0}^{2^{k'}}) \neq \star$ and k is the least such k';

2. the calling OTM goes undefined, if no such k' exists.[13]

The $\mathbf{0}^{2^k}$ values returned by calls to RUN are used as data for running our lower approximation of $\mathbf{q}(\overline{\kappa s}, |a|)$ in the same way we used the run times from calls to UNIV in Definition 11 as data for the running of our lower approximation of $\mathbf{q}(\overline{\Phi}_p, |a|)$. These clocked machines are perfectly free to query s outside of calls to RUN, but RUN is the only means of justifying big run times to the clocking mechanism. We call the class of functionals determined by such (extensional) machines the *clocked polynomial-time shred-operations.* Definition 21 in Section 6 provides the formal definitions. Note that a *clocked* polynomial-time shred-operation has domain $\mathcal{PR} \times \mathsf{N}$, whereas a polynomial-time shred-operation has just $\mathcal{R} \times \mathsf{N}$ as its domain; but a clocked polynomial-time shred-operation when restricted to $\mathcal{R} \times \mathsf{N}$ corresponds to some polynomial-time shred-operation.[14]

[12] Parameterizing these notions with respect to the class $\mathcal{S}$ is a bit irritating, but an analogous parameterization (with respect to the acceptable programming system φ) is implicit in the notion of effective operation.

[13] Note if $\mathsf{RUN}(x)$ returns $\langle s(x, \mathbf{0}^{2^k}), \mathbf{0}^{2^k}\rangle$, then $\kappa s(x) \leq 2^k < 2 \cdot \kappa s(x)$, and, hence, $s(x, \mathbf{0}^{2^k}) = \iota s(x)$. Also note that $\mathsf{RUN}(x)$ can be computed with only $1 + \log_2 \kappa s(x)$ calls to s. Moreover, assuming that, for all x, $\kappa s(x) \geq |x|$, the total time to compute $\mathsf{RUN}(x)$ is $\Theta(\kappa s(x) \log_2 \kappa s(x))$.

[14] Also note that this clocking scheme is based on *sequential* calls to RUN, and this causes problems for shred oracles outside of $\text{tot}(\mathcal{S}_{\text{all}})$; e.g., it is easy to show that $\mathbf{OR}'_{\|} = \lambda s, x.\mathbf{OR}_{\|}(\iota s, x)$ (where $\mathbf{OR}_{\|}$ is as in (1)) fails to be a clocked polynomial-time shred functional. We can rectify this problem by generalizing our clocking notion as follows. Replace the subroutine RUN above with a subroutine RACE that takes a nonempty list $\vec{x}$ of elements of N. A call to RACE on $\vec{x}$ results in: (i) $\langle s(x, \mathbf{0}^{2^k}), x, \mathbf{0}^{2^k}\rangle$, where k is the least number such that, for some x' in $\vec{x}$, $s(x', \mathbf{0}^{2^k}) \neq \star$ and x is the least such x'; or (ii) the calling OTM going undefined if no such k exists. Everything else can go as before. Clearly, $\mathbf{OR}'_{\|}$ can be computed by such clocked machines. We call the class of functionals determined by such (extensional) machines the *parallel-clocked polynomial-time shred-operations*. There turn out to be even more liberal notions of nonsequential polynomial-time shred functionals, but we do not consider these notions here.

5.2. Comparisons

The program behind our formalization of shreds and recursive shred-operations was: (i) to see if we could partially open up black boxes in some complexity-theoretically interesting fashion, (ii) to formalize a natural class of functionals based on these partially open black boxes that would be analogous to the polynomial-time operations, and (iii) to see if we could *provably* compare this new class of functionals to the basic polynomial-time functionals. Proposition 17 delivers this comparison.

Recall from Definition 15(g) that $\mathcal{S}_{\text{TM}}$ $\Big(= \{\, \chi_p : p \in \mathsf{N} \,\}\Big)$ denotes the collection of shreds based on Turing machine computations.

Proposition 17. *The following classes of functionals all correspond on $\mathcal{R}$:*

(a) The polynomial-time shred-operations with respect to $\mathcal{S}_{\text{TM}}$.

(b) The clocked polynomial-time shred-operations with respect to $\mathcal{S}_{\text{TM}}$.

(c) The basic polynomial-time functionals.[15]

The correspondence of (a) and (b) is the shred analog of Seth's Theorem 8 and the correspondence of (a) and (c) is the polynomial-time/shred analog of the Kreisel-Lacombe-Shoenfield Theorem (Theorem 3). Thus, one *can* partially open up black boxes and obtain something like the classical recursion theoretic correspondences of Theorems 2 and 3. If one replaces $\mathcal{S}_{\text{TM}}$ with either $\mathcal{S}_{\text{all}}$ or $\mathcal{S}_{\text{comp}} = \{\, s \in \mathcal{S}_{\text{all}} : s \text{ is computable} \,\}$ in Proposition 17, the analogous results are true and simpler to prove. However, consider M, an OTM computing a polynomial-time shred-operation with respect to $\mathcal{S}_{\text{TM}}$. M has as its oracle something that reasonably represents the computations of an *actual* TM program. Hence, the polynomial-time shred-operations with respect to $\mathcal{S}_{\text{TM}}$ correspond much more closely to polynomial-time operations than the polynomial-time shred-operations with respect to either $\mathcal{S}_{\text{all}}$ or $\mathcal{S}_{\text{comp}}$. There are difficulties with the use of $\mathcal{S}_{\text{TM}}$ in Proposition 17. The current proof of the proposition makes shameless use of special complexity properties of the TM model (see Lemma 18), and it is not clear how far the proposition generalizes to apply to a broad class of computation systems. Remark 22 in Section 6 discusses these problems in more detail.

6. Technical Details

The arguments below use standard techniques from elementary complexity theory and recursion theory. Familiarity with some standard theory of computation text (e.g., [HU79, DSW94]) should be sufficient background for these arguments.

[15] By Theorems 6 and 8, we can also add:

(d) The type-2 basic feasible functionals.

(e) The clocked basic polynomial-time functionals.

We can also add, but do not prove in this paper:

(f) The parallel-clocked polynomial-time shred-operations.

Background Results

Here we present a proof of Seth's Clocking Theorem, as ideas from this argument play an important role in the proof of Proposition 17 below. The proof of Theorem 8 is a considerable simplification of the one given in [Set92]. (Seth's original proof had other purposes beyond simply establishing Theorem 8.)

Theorem 8 (Seth's Clocking Theorem [Set92]). *The class of clocked basic polynomial-time functionals correspond on* $(\mathsf{N} \to \mathsf{N})$ *to the class basic polynomial-time functionals.*

Proof Sketch. We need to show that:

(a) Each $\mathbf{M}_{q,d}$ computes a basic polynomial-time functional.

(b) Each basic polynomial-time functional is computed by some $\mathbf{M}_{q,d}$.

Proof of (a). Clearly, $q^{[d]}(|f|, |a|)$ bounds the number of steps simulated by $\mathbf{M}_{q,d}$ on input (f, a). The overhead of the clocking machinations blows up the run time by no more than a quadratic amount. Hence there exists a constant c such that $c \cdot \left(q^{[d]}(|f|, |a|)\right)^2$ bounds the total run time of $\mathbf{M}_{q,d}$ on input (f, a). Therefore, (a) follows.

Proof of (b). Suppose $\mathbf{M}$ is an OTM that computes Γ with time bound given by $\mathbf{q}$, where $\mathbf{q}$ is a second-order polynomial over g and x. We may assume without loss of generality that $\mathbf{q} = q^{[d]}$ for some first-order polynomial q and $d \in \mathsf{N}$. We show that, for all $f : \mathsf{N} \to \mathsf{N}$ and all $a \in \mathsf{N}$, $\mathbf{M}(f, a) = \mathbf{M}_{q,d}(f, a)$. Fix f and a. Let t_* be the number of steps taken by $\mathbf{M}$ on input (f, a). By hypothesis, $t_* \leq \mathbf{q}(|f|, |a|)$. For each $t \leq t_*$, define

$$f_t \;=\; \lambda x.\begin{cases} f(x), & \text{if } \mathbf{M} \text{ on input } (f, a) \text{ makes the query} \\ & \quad f(x) = ? \text{ within its first } t \text{ steps;} \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

A straightforward induction argument shows that, for each $t \leq t_*$:

(i) After t steps, $\mathbf{M}_{q,d}$ on input (f, a) has $\mathbf{q}(|f_t|, |a|)$ as the contents of $\mathsf{x}[d]$.

(ii) $\mathbf{M}(f_t, a) = \mathbf{M}_{q,d}(f_t, a)$.

Hence $\mathbf{M}(f_{t_*}, a) = \mathbf{M}_{q,d}(f_{t_*}, a)$. But by (3) it follows that $\mathbf{M}(f, a) = \mathbf{M}(f_{t_*}, a)$. Hence, $\mathbf{M}(f, a) = \mathbf{M}_{q,d}(f, a)$, as claimed. Therefore, (b) follows. □

We state without proof the following lemma about Turing machines of which we make use.

Lemma 18 (The Patching Lemma). *For each φ-program p and for each finite function σ, there is another φ-program p_σ such that, for all x:*

$$\varphi_{p_\sigma}(x) \;=\; \begin{cases} \sigma(x), & \textit{if } x \in \text{domain}(\sigma); \\ \varphi_p(x), & \textit{otherwise.} \end{cases}$$

$$\Phi_{p_\sigma}(x) \;=\; \begin{cases} |x| + |\sigma(x)|, & \textit{if } x \in \text{domain}(\sigma); \\ \Phi_p(x), & \textit{otherwise.} \end{cases}$$

We also use the following lemma on second-order polynomials. The proofs of both parts of the lemma, which we omit, are simple inductions on the structure of the relevant second-order polynomials. *Notation:* If $f, g: \mathsf{N} \to \mathsf{N}$, then $f \leq g$ means that, for all x, $f(x) \leq g(x)$.

Lemma 19.

(a) For each second-order polynomial $\mathbf{q}$*, we have that, for all monotone nondecreasing* f, g: $\mathsf{N} \to \mathsf{N}$ *and all* $a, b \in \mathsf{N}$*,*

$$f \leq g \;\&\; a \leq b \implies \mathbf{q}(f, a) \leq \mathbf{q}(g, b).$$

(b) For all second-order polynomials $\mathbf{q}_1$ *and* $\mathbf{q}_2$*, there is another second-order polynomial* $\mathbf{q}_3$ *such that, for all monotone nondecreasing* $f: \mathsf{N} \to \mathsf{N}$ *and all* $a \in \mathsf{N}$*,*

$$\mathbf{q}_1(\lambda b\,.\,\mathbf{q}_2(f, b), a) \leq \mathbf{q}_3(f, a).$$

On Polynomial-Time Operations

The following proposition implies that, for any nontrivial, polynomial-time analog of effective operation, the "polynomial" upper bound of the cost of computing such thing needs to depend, in part, on the costs of running the program argument on various values during the course of the computation.

Proposition 9. *Suppose* i *is such that* φ_i *determines a total effective operation on* $\mathcal{R}$ *and that there is a second-order polynomial* $\mathbf{q}$ *such that, for all* p *with* φ_p *total and all* x*,* $\Phi_i(p, x)$ *runs within* $\mathbf{q}(|\varphi_p|, \max(|p|, |x|))$ *time. Then, there is a polynomial-time computable* $f: \mathsf{N} \to \mathsf{N}$ *such that, for all* p *with* φ_p *total and all* x*,* $\varphi_i(p, x) = f(x)$*.*

Proof. The argument is a variant of a standard proof of Rice's Theorem. (See Case's proof in either of [DW83, DSW94].) Suppose by way of contradiction that

(4) there are p_0, p_1, and x such that φ_{p_0} and φ_{p_1} are total and $\varphi_i(p_0, x) \neq \varphi_i(p_1, x)$.

If $\varphi_{p_0} = \varphi_{p_1}$, then clearly φ_i is not extensional in its first argument, a contradiction. So, suppose $\varphi_{p_0} \neq \varphi_{p_1}$. Let

$$g = \lambda n\,.\,\max(|\varphi_{p_0}|(n), |\varphi_{p_1}|(n)).$$

By the recursion theorem (see [Rog67, Exercise 11-4] or [RC94]), there is a φ-program e such that, for all y,

$$\varphi_e(y) = \begin{cases} 0, & \text{if (i): } \Phi_i(e, x) > \mathbf{q}(g, \max(|e|, |x|)); \\ \varphi_{p_1}(y), & \text{if (ii): } \textit{not} \text{ (i) and } \varphi_i(e, x) = \varphi_i(p_0, x); \\ \varphi_{p_0}(y), & \text{if (iii): otherwise.} \end{cases} \tag{5}$$

Note that the clauses (i), (ii), and (iii) in (5) do not depend on y. Also note that, whichever of clauses (i), (ii), and (iii) hold, φ_e is total, and hence $\varphi_i(e, x)\!\downarrow$. We consider the following three exhaustive cases.

Case 1: Clause (i) in (5) holds. Then, $\varphi_e = \lambda y\,.\,0$. Hence by our hypotheses on i, $\Phi_i(e,x) \leq \mathbf{q}(|\varphi_e|, \max(|e|,|x|))$. By Lemma 19(a), we have $\mathbf{q}(|\varphi_e|, \max(|e|,|x|)) \leq \mathbf{q}(g, \max(|e|,|x|))$. Hence, $\Phi_i(e,x) \leq \mathbf{q}(g, \max(|e|,|x|))$, which contradicts clause (i).

Case 2: Clause (ii) in (5) holds. Then, $\varphi_e = \varphi_{p_1}$, hence $\varphi_i(e,x) = \varphi_i(p_1,x)$, by φ_i's extensionality. But, since $\varphi_i(p_1,x) \neq \varphi_i(p_0,x)$, this contradicts clause (ii).

Case 3: Clause (iii) in (5) holds. Then, $\varphi_e = \varphi_{p_0}$, hence $\varphi_i(e,x) = \varphi_i(p_0,x)$, by φ_i's extensionality. But, in this case, clause (ii) should hold, which contradicts clause (iii).

Thus, since (4) fails, we have that, for all p_0 and p_1 with φ_{p_0} and φ_{p_1} total and all x, $\varphi_i(p_0,x) = \varphi_i(p_1,x)$. Let $p_\star$ be a φ-program for $\lambda x\,.\,0$. Then $f = \lambda x\,.\,\varphi_i(p_\star,x)$ is as required. □

Recall from Section 4.2 that, for each $\alpha\colon \mathsf{N} \rightharpoonup \mathsf{N}$ and $x \in \mathsf{N}$,

$$\Gamma_0(\alpha,x) \;=_{\text{def}}\; \begin{cases} \uparrow, & \text{if (i): for some } y \in \{\mathbf{0},\mathbf{1}\}^{|x|},\ \alpha(y)\uparrow; \\ 1, & \text{if (ii): } \textit{not} \text{ (i) and } (\exists y \in \{\mathbf{0},\mathbf{1}\}^{|x|})[\,\alpha(y) \text{ is odd }]; \\ 0, & \text{if (iii): } \textit{not} \text{ (i) and } (\forall y \in \{\mathbf{0},\mathbf{1}\}^{|x|})[\,\alpha(y) \text{ is even }]. \end{cases}$$

Proposition 12.

(a) The restriction of Γ_0 *to* $(\mathsf{N} \to \mathsf{N}) \times \mathsf{N}$ *is not basic feasible. Moreover, there is an honest, exponential-time computable function* g *such that, for each* q *and* d*, there is an* n *for which* $\Gamma_0(g, \mathbf{0}^n) \neq \mathbf{M}_{q,d}(g, \mathbf{0}^n)$.

(b) If $\mathrm{P} = \mathrm{NP}$*, then* Γ_0 *is a clocked polynomial-time operation.*

Proof of Proposition 12. The proof of part (a) is a standard oracle construction where, in this case, g is the oracle constructed.

For part (b), first suppose $\mathrm{P} = \mathrm{NP}$. Then consider the predicates:

$$P(p,\mathbf{0}^m,\mathbf{0}^n) \;\equiv\; \big(\exists x \in \{\mathbf{0},\mathbf{1}\}^m\big)\big[\,\Phi_p(x) > n\,\big].$$

$$Q(p,\mathbf{0}^n,x_0,x_1) \;\equiv\; (\exists x : x_0 \leq x \leq x_1)\big[\,\Phi_p(x) > n\,\big].$$

$$R(p,\mathbf{0}^m,\mathbf{0}^n) \;\equiv\; \big(\exists x \in \{\mathbf{0},\mathbf{1}\}^m\big)\big[\,\Phi_p(x) \leq n \text{ and } \varphi_p(x) \text{ is odd}\,\big].$$

Clearly, P, Q, and R each are nondeterministically decidable in time polynomial in the lengths of their arguments. Hence, since $\mathrm{P} = \mathrm{NP}$, P, Q, and R are each in polynomial-time. Fix polynomial-time decision procedures for P, Q, and R, and let q_1 be a polynomial such that, for all arguments, the run times of these procedures are less than q_1(the sum of the lengths of the arguments). Let ψ be the partial recursive function computed by the following informally stated program.

Program for ψ

Input p, x.

Set $m \leftarrow |x|$ and $n \leftarrow \max(|p|, m)$.

While $P(p,\mathbf{0}^m,\mathbf{0}^n)$ **do**

Use Q in a binary search to find an $x_0 \in \{\mathbf{0}, \mathbf{1}\}^m$ such that $\Phi_p(x_0) > n$.

Set $n \leftarrow 2 \cdot \Phi_p(x_0)$. *(Note: If $\Phi_p(x_0)\uparrow$, then the computation diverges.)*

End while

If $R(p, \mathbf{0}^m, \mathbf{0}^n)$ then output 1 else output 0.

End program

Clearly, $\psi = \lambda p, x\,.\,\Gamma_0(\varphi_p, x)$. We argue that one can insert an appropriate clocking mechanism into the above program so as to make it equivalent to an $M_{q,d}$. Note that throughout the course of execution of the program that $n \geq \max(|p|, m)$. Now, evaluating $P(p, \mathbf{0}^m, \mathbf{0}^n)$ in the **while** test takes $q_1(n)$ time, and using Q in the binary search takes $c \cdot n \cdot q_1(n)$ time for some constant c. Determining $\Phi_p(x_0)$ can be done through a normal query to UNIVand, once we know the value of $\Phi_p(x_0)$, we can bound the cost of the next iteration by $q_2(\Phi_p(x_0))$, where q_2 is an appropriate polynomial such that, for all n, $q_2(n) > c \cdot (2n+2) \cdot q_1(2n)$. Thus, with appropriate choice of q, it is clear that we can transform the above program into an equivalent $M_{q,1}$. Hence, part (b) follows. □

Remark 20. The only use of the P = NP hypothesis in the argument for Proposition 12(b) is in making the predicates P, Q, and R polynomial-time decidable. One can exploit this to convert the argument into a construction of an oracle relative to which: (i) Γ_0 is again a clocked polynomial-time operation, and (ii) P ≠ NP. Hence P = NP is not equivalent to the failure of the correspondence on $\mathcal{R}$ of the clocked polynomial-time operations the basic polynomial-time functionals. ◇

Proposition 13. *If* P = NP*, then the polynomial-time operations correspond on $\mathcal{R}$ to the clocked polynomial-time operations.*

Proof. Suppose M determines a total polynomial-time operation on $\mathcal{R}$ and $\mathbf{q}$ is a second-order polynomial such that, for all p with φ_p total and all x, $M(p, x)$ runs in time $\mathbf{q}(\overline{\Phi}_p, |x|)$. Without loss of generality, suppose $\mathbf{q} = q^{[k]}$ for some polynomial q and $k \in \mathsf{N}$. Using the P = NP hypothesis and the technique of the proof of Proposition 12, construct a TM M' that (i) clockably computes $\mathbf{q}(\overline{\Phi}_p, |x|)$ and then (ii) runs M on p and x. A straightforward argument shows that M' corresponds to a $M_{q,d}$. □

On Polynomial-Time Shred-Operations

Recall that M ranges over OTMs whose function oracle, s, is in $\mathcal{S}_{\text{all}}$.

Definition 21.

(a) Let RUN denote a fixed OTM subroutine such that when an OTM, M calls RUN on $x \in \mathsf{N}$, the result is

(i) $\langle s(x, \mathbf{0}^{2^k}), \mathbf{0}^{2^k}\rangle$ is returned, if k is the least number such that $s(x, \mathbf{0}^{2^k}) \neq \star$; and

(ii) M goes undefined, if no such k exists.

(b) A *special oracle Turing machine* (SOTM) M is an oracle Turing machine defined, as follows. M takes an oracle $s \in \mathcal{S}_{\text{all}}$ and an input $x \in \mathsf{N}$. M includes RUN as a subroutine and obeys the same constraints M (of Definition 11) does with respect to UNIV.

(c) Suppose M is an SOTM, q is a polynomial over two variables, and $d \in \mathsf{N}$. Let $\mathsf{M}_{q,d}$ be the SOTM that, on input (s, a), operates as follows. $\mathsf{M}_{q,d}$ maintains a counter clock and two arrays x[0..d] and y[0..$d-1$]. $\mathsf{M}_{q,d}$ maintains the following invariants, where $i = 0, \ldots, d-1$.

$$\begin{aligned} \mathsf{x}[0] &= q(0, |a|). \\ \mathsf{x}[i+1] &= q(\mathsf{y}[i], |a|). \\ \mathsf{y}[i] &= \max\left(\left\{ \mathbf{0}^{2^k} : \begin{array}{l} |x| \leq \mathsf{x}[i] \text{ and the call RUN } (x) \text{ was} \\ \text{made and returned } \langle s(x, \mathbf{0}^{2^k}), \mathbf{0}^{2^k}\rangle \end{array} \right\}\right). \end{aligned}$$

On start up, $\mathsf{M}_{q,d}$ initializes clock, x, and y exactly as $\mathbf{M}_{q,d}$ does. Then, $\mathsf{M}_{q,d}$ simulates M step-by-step on input (s, a). For each step of M simulated:

- If the step of M just simulated was the last step of an execution of RUN, then, if necessary, $\mathsf{M}_{q,d}$ recomputes the x[i]'s and y[i]'s to re-establish the invariants.
- If, in the step of M just simulated, M halts with output y, then $\mathsf{M}_{q,d}$ outputs y and halts.
- If M did not halt in the step just simulated, then the value of clock is increased by 1 and, if clock $<$ x[d] *or* if the step of M just simulated was part of an execution of RUN, then the simulation continues; otherwise, $\mathsf{M}_{q,d}$ outputs 0 and halts.

(d) Suppose $\mathcal{S} \subseteq \mathcal{S}_{\text{all}}$ is such that $\mathcal{PR} = \iota\mathcal{S}$. $\Gamma\colon \mathcal{PR} \times \mathsf{N} \rightharpoonup \mathsf{N}$ is a *clocked polynomial-time shred-operation* with respect to $\mathcal{S}$ if and only if there exists an extensional (with respect to $\mathcal{S}$) $\mathsf{M}_{q,d}$ that determines Γ as per Definition 16. ◇

Proposition 17. *The following classes of functionals all correspond on $\mathcal{R}$:*

(a) The polynomial-time shred-operations with respect to $\mathcal{S}_{\text{TM}}$.

(b) The clocked polynomial-time shred-operations with respect to $\mathcal{S}_{\text{TM}}$.

(c) The basic polynomial-time functionals.

Proof. *Convention:* All clocked and unclocked polynomial-time shred-operations mentioned in this proof will be with respect to $\mathcal{S}_{\text{TM}}$. So, to cut the clutter, the "with respect to $\mathcal{S}_{\text{TM}}$" clause will be dropped below.

When all the functionals concerned are restricted to $\mathcal{R}$ we clearly have that

$$\text{(a)} \supseteq \text{(b)} \supseteq \text{(c)}.$$

To establish the reverse containments, we show that (a) $\subseteq$ (c) by an argument that borrows ideas from the proof given for Theorem 8.

Suppose $\Gamma\colon \mathcal{R} \times \mathsf{N} \to \mathsf{N}$ is a polynomial-time shred-operation. Suppose also that M, an OTM, and $\mathbf{q}$, a second-order polynomial, are such that, for all $s \in \mathcal{S}_{\mathrm{TM}}$ and $a \in \mathsf{N}$,

1. $\mathsf{M}(s, a) = \Gamma(\iota s, a)$, and
2. M on (s, a) runs within $\mathbf{q}(\overline{\kappa s}, |a|)$ time.

By Lemma 19(b), there is a second-order polynomial $\mathbf{q}_0$ such that, for all $g\colon \mathsf{N} \to \mathsf{N}$ and all $a \in \mathsf{N}$,

$$\mathbf{q}(\lambda n\,.\,(n + |g|(n)), |a|) \quad \leq \quad \mathbf{q}_0(|g|, |a|). \tag{6}$$

Claim 1. *For each $g \in \mathcal{R}$ and $a \in \mathsf{N}$, there is a φ-program $p_{g,a}$ such that*

(a) $\varphi_{p_{g,a}} = g$,

(b) for all x with $|x| \leq \mathbf{q}_0(|g|, |a|)$, $\Phi_{p_{g,a}}(x) = |x| + |g(x)|$, and

(c) M on $(\chi_{p_{g,a}}, a)$ runs within $\mathbf{q}_0(|g|, |a|)$ time.

Proof. Fix a and g. Let σ be the finite function with the graph $\{\, (x, g(x)) : |x| \leq \mathbf{q}_0(|g|, |a|) \,\}$. Fix p, some φ-program for g. By Lemma 18 there is a φ-program $\widehat{p}$ for g such that, for all x,

$$\Phi_{\widehat{p}}(x) \quad = \quad \begin{cases} |x| + |\sigma(x)|, & \text{if } x \in \mathrm{domain}(\sigma); \\ \Phi_p(x), & \text{otherwise.} \end{cases}$$

We argue that taking $\widehat{p}$ for $p_{g,a}$ suffices for the claim. Parts (a) and (b) are clearly satisfied. It remains to show part (c). Let p_0 be a φ-program such that $\varphi_{p_0} = \lambda x\,.\,0$ and $\Phi_{p_0} = \lambda x\,.\,|x|$. By Lemma 18 again, there is a φ-program $\widehat{p}_0$ such that, for all x:

$$\varphi_{\widehat{p}_0}(x) \quad = \quad \begin{cases} \sigma(x), & \text{if } x \in \mathrm{domain}(\sigma); \\ 0, & \text{otherwise.} \end{cases}$$

$$\Phi_{\widehat{p}_0}(x) \quad = \quad \begin{cases} |x| + |\sigma(x)|, & \text{if } x \in \mathrm{domain}(\sigma); \\ |x|, & \text{otherwise.} \end{cases} \tag{7}$$

By our hypotheses on M, M on $(\chi_{\widehat{p}_0}, a)$ runs within $\mathbf{q}(\overline{\kappa\chi_{\widehat{p}_0}}, |a|)$ time. Note that

$$\begin{aligned} \mathbf{q}(\overline{\kappa\chi_{\widehat{p}_0}}, |a|) \quad &= \quad \mathbf{q}(\overline{\Phi_{\widehat{p}_0}}, |a|) && \text{(by Definition 15(c))} \\ &\leq \quad \mathbf{q}\,(\lambda n\,.\,(n + |g|(n)), |a|) && \text{(by (7) and Lemma 19(a))} \\ &\leq \quad \mathbf{q}_0(|g|, |a|) && \text{(by (6)).} \end{aligned}$$

Hence, M on $(\chi_{\widehat{p}_0}, a)$ runs within $\mathbf{q}_0(|g|, |a|)$ time. So, in the course of M's computation on $(\chi_{\widehat{p}_0}, |a|)$, all of M's queries of the form $s(x, \mathbf{0}^k) =?$ involve x's with $|x| \leq \mathbf{q}_0(|g|, |a|)$, i.e., x's in the domain of σ. Since for $x \in \mathrm{domain}(\sigma)$ we have both $\varphi_{\widehat{p}_0}(x) = \varphi_{\widehat{p}}(x)$ and $\Phi_{\widehat{p}_0}(x) = \Phi_{\widehat{p}}(x)$, it follows that M's computations on $(\chi_{\widehat{p}_0}, a)$ and $(\chi_{\widehat{p}}, a)$ are identical. Thus, $\mathsf{M}(\chi_{\widehat{p}_0}, a) = \mathsf{M}(\chi_{\widehat{p}}, a)$. Since $\iota\chi_{\widehat{p}} = g$, we have by M's extensionality that $\mathsf{M}(\chi_{\widehat{p}_0}, a) = \mathsf{M}(\chi_{\widehat{p}}, a) = \Gamma(g, a)$. Therefore, part (c) follows. □ **Claim 1**

Consider the OTM **M** whose program is sketched below.

Program for M

Input a with oracle g.

Go through a step-by-step simulation M on input a. Each M step that is *not* an oracle call is faithfully carried out. Each oracle query, $s(x, \mathbf{0}^k) =?$, is simulated as follows.

Condition 1. $k < |x| + |g(x)|$.

Make $\star$ the answer to the query in the simulation of M.

Condition 2. $k \geq |x| + |g(x)|$.

Give $g(x)$ as the answer to M's query.

If, in the course of the simulation, M halts with output y, then output y and halt.

End program

Claim 2.

(a) There is a second-order polynomial that, for all $g\colon \mathsf{N} \to \mathsf{N}$ *and* $a \in \mathsf{N}$*, bounds the run time of* $\mathbf{M}$ *on* (g, a)*.*

(b) For all $g \in \mathcal{R}$ *and* $a \in \mathsf{N}$, $\mathbf{M}(g, a) = \Gamma(g, a)$*.*

Proof. Fix $g \in \mathcal{R}$ and $a \in \mathsf{N}$ and let $p_{g,a}$ be as in Claim 1. Consider the computation of M on $(\chi_{p_{g,a}}, a)$. It follows by an induction on the steps of this computation that these steps are identical to the steps of M simulated by $\mathbf{M}$ on (g, a). Therefore, $\mathbf{M}(g, a) = \mathsf{M}(\chi_{p_{g,a}}, a) = \Gamma(\iota\chi_{p_{g,a}}, a) = \Gamma(g, a)$ and $\mathbf{M}$ on (g, a) simulates no more than $\mathbf{q}_0(|g|, |a|)$ many steps of M. By a straightforward argument there are constants c and m, independent of g and a, such that on (g, a), $\mathbf{M}$ runs within $c \cdot (\mathbf{q}_0(|g|, |a|))^m$ time.

Fix $g \notin \mathcal{R}$ and $a \in \mathsf{N}$ and let $g' \in \mathcal{R}$ be such that, for all x with $|x| \leq \mathbf{q}_0(|g|, |a|)$, $g(x) = g'(x)$. By an argument similar to that of the previous paragraph we have that $\mathbf{M}(g', a) = \mathbf{M}(g, a)$ and $\mathbf{M}$ on (g, a) runs within $c \cdot (\mathbf{q}_0(|g'|, |a|))^m = c \cdot (\mathbf{q}_0(|g|, |a|))^m$ time, where c and m are as before.

Hence, parts (a) and (b) of the claim follow. □ **Claim 2**

Therefore, $\mathbf{M}$ determines a basic polynomial-time functional that corresponds with Γ on $\mathcal{R}$. Hence, (a) $\subseteq$ (c). □

Remark 22. The strong dependence on Lemma 18 in the above argument is unsatisfying, but it is indicative of deeper problems. Consider an acceptable programming system φ' and associated complexity measure Φ' that are polynomially related to our standard, Turing machine-based φ and Φ, but that are such that, for each p with $\varphi'_p(0)\!\downarrow$, one can somehow effectively reconstruct p from $\Phi'_p(0)$. Let χ' be the computation system associated with the φ' and Φ'. Any attempt to prove the analog of Proposition 17 will run into the difficulties of Section 4. What is probably needed for the analog of Proposition 17 to be true for a given computation system χ'' is some strong, complexity theoretic version of Rice's Theorem to hold for the φ'' and Φ'' with which χ'' is associated. ◇

7. Further Problems

The results of Section 4 indicate that the original question of whether a polynomial-time analog of the Kreisel-Lacombe-Shoenfield Theorem holds seems to be, like P = NP?, yet another technically intractable complexity theoretic problem. How important a problem this is, I can't say. Some of the key problems in contemporary programming languages center around the issue of information hiding, e.g., data structures that hide their implementations. My guess is that some of these programming language problems can be sharpened to the point where they become interesting complexity theoretic questions, and in such a context the polynomial-time Kreisel-Lacombe-Shoenfield problem may play an interesting role.

Section 5 showed that by weakening the notion of effective operation one can obtain a polymial-time analog of the Kreisel-Lacombe-Shoenfield Theorem. One obvious question left open is whether shreds can be replaced with real traces and still obtain the equivalence. My guess is "yes." Computations can be very coy about what they are up to until very late in their course, e.g., they can run lots of unrelated subcomputations and leave until the very end which of these subcomputations are used to produce the final result of the main computation.

In the theory of programming languages, the effectively continuous functionals (Definition 1(c)) and their generalizations play a much greater role than the partial recursive functionals (Definition 1(b)). So, another set of problems concerns the parallel-clocked polynomial-time shred-operations of footnote 14. These functionals in some respects resemble the effective continuous functionals. How close is this resemblance? Can one obtain a language characterization of this class along the lines of Cook and Kapron's characterizations of the basic feasible functionals [CK90] or of Plotkin's PCF [Plo77]? As noted in footnote 14, there is an even more general class of "polynomial-time shred-operations" on $\mathcal{PR} \times \mathsf{N}$. Is there a most general "polynomial-time shred-operation" and, if so, can one prove some analog of the Myhill-Shepherdson Theorem for this class?

I am curious to see if the ideas and results presented above are useful in extending type-2 complexity beyond (and below) polynomial-time to develop a general, machine-based theory of type-2 computational complexity. Additionally, I am hopeful that shreds, or something like them, will be of help in sorting out useful machine models for computation at above type 2. Functional programming techniques like continuations and monads are naturally set at type 3. It would be great fun to have good type-3 machine models so as to subject algorithms built through such techniques to complexity analyses.[16]

[16] Recently Seth [Set94, Set95] gave an extension of the Kapron and Cook Theorem (Theorem 6 above) to all finite types.

References

[BDG90] J. Balcázar, J. Díaz, and J. Gabarró, *Structural complexity II*, Springer-Verlag, 1990.

[Boo74] R. Book, *Tally languages and complexity classes*, Information and Control **26** (1974), 186–193.

[Bus86] S. Buss, *The polynomial hierarchy and intuitionistic bounded arithmetic*, Structure in Complexity Theory (A. Selman, ed.), Springer-Verlag, 1986, pp. 77–103.

[CK90] S. Cook and B. Kapron, *Characterizations of the basic feasible functions of finite type*, Feasible Mathematics: A Mathematical Sciences Institute Workshop, (S. Buss and P. Scott, eds.), Birkhäuser, 1990, pp. 71–95.

[Cob65] A. Cobham, *The intrinsic computational difficulty of functions*, Proc. Int. Conf. Logic, Methodology and Philosophy (Y. Bar Hillel, ed.), North-Holland, 1965, pp. 24–30.

[Coo91] S. Cook, *Computability and complexity of higher type functions*, Logic from Computer Science (Y.N. Moschovakis, ed.), Springer-Verlag, 1991, pp. 51–72.

[CU89] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Proc. of the 21st Ann. ACM Symp. on Theory of Computing, 1989, pp. 107–112.

[CU93] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Annals of Pure and Applied Logic **63** (1993), 103–200.

[DSW94] M. Davis, R. Sigal, and E. Weyuker, *Computability, complexity, and languages*, Academic Press, 1994, second edition of [DW83].

[DW83] M. Davis and E. Weyuker, *Computability, complexity, and languages*, Academic Press, 1983.

[Fri58] R. Friedberg, *Un contre-example relatif aux fonctionnelles récursives*, Comptes Rendus Hebdomadaires des séances de l'Académie des Sciences **247** (1958), 852–854.

[HU79] J. Hopcroft and J. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.

[KC91] B. Kapron and S. Cook, *A new characterization of Mehlhorn's polynomial time functionals*, Proc. of the 32nd Ann. IEEE Symp. Found. of Comp. Sci., 1991, pp. 342–347.

[KC96] B. Kapron and S. Cook, *A new characterization of type 2 feasibility*, SIAM Journal on Computing **25** (1996), 117–132.

[KLS57] G. Kreisel, D. Lacombe, and J. Shoenfield, *Partial recursive functionals and effective operations*, Constructivity in Mathematics: Proceedings of the Colloquium held at Amsterdam (A. Heyting, ed.), North-Holland, 1957, pp. 195–207.

[Meh76] K. Mehlhorn, *Polynomial and abstract subrecursive classes*, Journal of Computer and System Science **12** (1976), 147–178.

[MS55] J. Myhill and J. Shepherdson, *Effective operations on partial recursive functions*, Zeitschrift für Mathematische Logik und Grundlagen der Mathematik **1** (1955), 310–317.

[MY78] M. Machtey and P. Young, *An introduction to the general theory of algorithms*, North-Holland, 1978.

[Odi89] P. Odifreddi, *Classical recursion theory*, North-Holland, 1989.

[Plo77] G. Plotkin, *LCF considered as a programming language*, Theoretical Computer Science **5** (1977), 223–255.

[RC94] J. Royer and J. Case, *Subrecursive programming systems: Complexity & succinctness*, Birkhäuser, 1994.

[Rog67] H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967, reprinted, MIT Press, 1987.

[Sco75] D. Scott, *Lambda calculus and recursion theory*, Proceedings of the Third Scandinavian Logic Symposium (S. Kanger, ed.), North-Holland, 1975, pp. 154–193.

[Set92] A. Seth, *There is no recursive axiomatization for feasible functionals of type 2*, Seventh Annual IEEE Symposium on Logic in Computer Science, 1992, pp. 286–295.

[Set94] A. Seth, *Complexity theory of higher type functionals*, Ph.D. thesis, University of Bombay, 1994.

[Set95] A. Seth, *Turing machine characterizations of feasible functionals of all finite types*, Feasible Mathematics II (P. Clote and J. Remmel, eds.), Birkhauser, 1995, pp. 407–428.