Characterizations of the Basic Feasible Functionals of Finite Type

Extended Abstract

Stephen A. Cook and Bruce M. Kapron
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

1 Introduction

Functionals are functions which take natural numbers and other functionals as arguments and return natural numbers as values. The class of "feasible" functionals of finite type was introduced in [6] via the typed lambda calculus, and used to interpret certain formal systems of arithmetic: systems capturing the notion of "feasibly constructive proof" (we equate feasibility with polynomial time computability). Here we name the functionals of [6] the *basic feasible functionals* and justify the designation by presenting results which include two programming style characterizations of the class. We also give examples of both feasible and infeasible functionals, and argue that the notion plays a natural role in complexity theory.

Type 2 functionals take numbers and ordinary numerical functions as arguments. When these argument functions are 0-1 valued (i.e. sets) the notion of polynomial time reducibility introduced by Cook [5] provides a natural definition for polynomial time functional in which the input sets are accessed by a Turing machine using oracles. This definition was generalized (in different ways) by Constable [4] and Mehlhorn [16] to capture the class of "polynomial operators", which allow more general functions $f : \mathbb{N} \to \mathbb{N}$ as arguments. (Mehlhorn's class is essentially our type 2 basic feasible functionals.)

For type 3 and above, we are not aware of any attempts to define polynomial time objects prior to [6]. On the other hand Kleene [12] and [13] introduced and studied the *recursive* functionals of all finite types. Both Kleene [14], [15] and Gandy [8] subsequently gave machine or programming style characterizations of these functionals, but the infinitary nature of these characterizations makes defining run times problematic. Hence the work of Kleene and Gandy does not immediately point to a definition of polynomial time for functionals of type 3 and above.

For the present paper we define a simple typed while-programming language which generalizes the sort of simple language used in computability texts to define the familiar numerical computable functions, and which corresponds roughly to the μ-recursion of [12], §8.1. This language does not fully capture the notion of higher type computability, as explained in section 4. We define run times for our programs and prove in the style of Ritchie [20] and Cobham [3] that the feasible functionals of [6] are precisely those functionals computable by typed while-programs with run times feasibly length-bounded (we say that b is a *length-bound* for a if $a \leq |b|$, where $|b| = \lfloor \log_2(b+1) \rfloor$).

For our second characterization we introduce the notion of a bounded typed loop program, which generalizes a simple bounded loop programming language for the familiar type 1 polynomial time functions. We prove that a finite type functional is feasible iff it is computable by a bounded typed loop program.

2 Examples

A type 0 functional is just a natural number. A type 1 functional is function $f : \mathbb{N}^k \to \mathbb{N}$ for some $k \geq 1$. In general, a type $t+1$ functional takes as arguments a type t functional, together possibly with functionals of types $\leq t$, and returns a natural number value.

In constructive analysis [1], a real number is presented by a Cauchy sequence of rationals with a certain convergence property. Thus a real function can be considered a type 2 functional in our sense, although feasible real functions can be presented as type 1 functions (see [10]). Integration can then be viewed as either a type 3 or type 2 operator. If the operator is restricted to feasible real functions, it becomes computable but not feasible [7]. On the other hand, the functional which takes feasible real contractors to their fixed points appears to be feasible ([10], Thm 3.6).

Luby and Impagliazzo [11] recently showed how to transform a one way function to a pseudo-random number generator. This transformation can be viewed in a natural way as a type 2 feasible functional. Other constructions in feasible cryptography theory have a similar interpretation.

Lower bounds in the decision tree model for sorting and selection can be derived by counting arguments and by adversary arguments. Although the decision tree may have size exponential in the number n of input objects, if the tree is presented locally by an oracle, an adversary argument will usually produce a "bad" path in polynomial time given a tree that is too small. Thus Rackoff [19], suggested using the notion of feasible functional to distinguish between adversary and counting arguments.

A notion of realizability in constructive logic can be briefly explained as follows. A formula $\forall x \exists y A(x, y)$ of first order arithmetic is *realized* by a function f if $\forall x A(x, f(x))$ holds. A functional Φ realizes a formula $(A \supset B)$ if for each functional F realizing A, $\Phi(F)$ realizes B. Now consider the induction axiom

2.1 $\quad [B(0) \wedge \forall x(B(x) \supset B(x+1))] \supset \forall x B(x).$

If $B(x)$ is $\forall y \exists z A(x, y, z)$, then $B(x)$, if it is true, is realized by a type 1 function f above, and $\forall x(B(x) \supset B(x+1))$ by a type 2 functional (if true). Thus the induction axiom itself (which is always true) is realized by a type 3 functional Φ. The functional Φ is computable for each formula $B(x)$, but Φ is not in general feasible. If $B(x)$ is $\exists z \leq y A(x, y, z)$, with $A(x, y, z)$ a feasible predicate, and if the induction axiom (2.1) above is replaced by "induction on notation", then the realizing functional is feasible [6].

In section 6 we present a basic feasible type 3 functional Φ_1 which is "discontinuous" in the sense that for some type 2 functional F, the value of F is required at infinitely many arguments to determine $\Phi_1(F)$. We also present a simple variation Φ_2 of Φ_1 and give an interesting proof that Φ_2 is not basic feasible.

3 The Basic Feasible Functionals

The set of *types* is defined inductively as follows:

- 0 is a type
- $(\sigma \rightarrow \tau)$ is a type, if σ and τ are types.

The set $Fn(\tau)$ of *functionals of type* τ is defined by induction on τ:

- $Fn(0) = \mathbb{N}$
- $Fn(\sigma \rightarrow \tau) = \{F | F : Fn(\sigma) \rightarrow Fn(\tau)\}$.

It is not hard to show that each type τ has a unique normal form

3.1 $\quad \tau = \tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_k \rightarrow 0$

where the missing parentheses are put in with association to the right. Hence a functional F of type τ is considered in a natural way as a function of variables $X_1, \ldots, X_k$, with X_i ranging over $Fn(\tau_i)$, and returning a natural number value:

$$F(X_1)(X_2)\ldots(X_k) = F(X_1, \ldots, X_k).$$

The type 1 feasible functionals are the familiar polynomial time computable functions (each $\tau_i = 0$ in 3.1). The type 2 feasible functionals can be easily defined by polynomial time Turing machines with oracles for the input functions, and a cost of one time step per query, provided that the input functions are 0-1 valued (i.e. sets), or at least polynomially bounded. If the input functions grow fast, then we allow the Turing machine to run in time polynomial in both the lengths of the values returned by a bounded number of oracle calls and the lengths of the type 0 inputs. For example, the functional *Apply* defined by

$$Apply(X, y) = X(y)$$

(where X has type $0 \rightarrow 0$ and y has type 0) is feasible in our sense, even though the input function X may grow arbitrarily rapidly.

That the functional *Apply* should be feasible may not be intuitively obvious. For example, in the system HRO (hereditarily recursive operations: see [21]) the input functionals must be computable, and they are represented by Gödel numbers of machines computing them. Thus there are no input oracles, and to compute $Apply(X, y)$ a machine must simulate the machine computing X on input y. This operation is clearly not feasible in the length of the Gödel number for X. To get around this difficulty, Buss [2] introduced a complex type theory in which types are recursively labelled with run time bounds (see also [17]). The use of oracles leads to a much simpler theory.

The inputs to type 3 functionals are type 2 functionals, and the oracles for these inputs themselves take type 1 functions as arguments. These type 1 functions can be described by subprograms. Because of this subprogram structure, it appears that programming languages rather than Turing machines are a more natural setting for our investigation. This approach is carried out in sections 4 and 5. Here we repeat the original definition [6], in which the functionals in question are those definable by terms in an equational system called PV^ω. These terms are expressions in the typed λ-calculus, including a recursor.

The PV^ω terms (or just *terms*) are defined inductively as follows:

- For each type σ there are infinitely many variables $X^\sigma, Y^\sigma, Z^\sigma, \ldots$ of type σ, and each such variable is a term of type σ.
- For each type 1 type $\sigma = 0 \to 0 \to \cdots \to 0$ and each polynomial time computable function of type σ there is a function symbol f^σ representing that function. f^σ is a term of type σ.
- The type 2 recursor $\mathcal{R}$ is a term (the type is explained below).
- If T is a term of type τ and X is a variable of type σ, then $(\lambda X.T)$ is a term of type $(\sigma \to \tau)$ (an abstraction).
- If S is a term of type $(\sigma \to \tau)$ and T is a term of type σ, then (ST) is a term of type τ (an application).

For readability, we will often write $S(T)$ for (ST); we will also write $S(T_1, \ldots, T_k)$ for $(\ldots((ST_1)T_2)\ldots T_k)$, and write $\lambda X_1 \ldots X_k.T$ or $\lambda \vec{X}.T$ for $(\lambda X_1.(\lambda X_2.(\ldots(\lambda X_k.T)\ldots)))$.

The recursor $\mathcal{R}$ represents higher type limited recursion on (binary) notation, which generalizes Cobham's limited recursion on notation. It takes arguments y, Z, W, x of types 0, $0 \to 0 \to 0$, $0 \to 0$, and 0 respectively, and returns a value of type 0. Here $Z(x,u)$ is the next step function and $W(x)$ is the bounding function. The meaning of $\mathcal{R}$ is given by the equation:

3.2 $$\mathcal{R}(y, Z, W, x) = \begin{cases} \text{if } x = 0 \text{ then } y \\ \text{else if } |t| \leq |W(x)| \text{ then } t\ , \\ \text{else } W(x) \end{cases}$$

where t is the term

$$Z(x, \mathcal{R}(y, Z, W, \lfloor \tfrac{1}{2} x \rfloor)).$$

We note that if fixed polynomial time computable functions are substituted for Z and W, then 3.2 defines a polynomial time computable function of y and x.

In a natural way (which we will spell out in the full paper), each closed (i.e. having no free variables) feasible λ-term T of type τ represents a functional in $Fn(\tau)$.

3.3 Definition *A functional is* basic feasible *if some closed PV^ω term represents it.*

For example, the functinal *Apply* is represented by the term $\lambda Xy.X(y)$ (which is formally $(\lambda X.(\lambda y.Xy))$). A more interesting example is the functional $Smax$, of type $(0 \to 0) \to 0 \to 0$, defined by

3.4 $$Smax(f, n) = \max_{k \subseteq n} f(k),$$

where we write $k \subseteq n$ to denote that the binary representation of k is an initial segment of the binary representation of n (as strings). Since $Smax(f,n) = f(T(f,n))$, where

$$T(f,n) = \begin{cases} \text{if } n = 0 \text{ then } 0 \\ \text{else if } f(x) > f(T(f, \lfloor \frac{n}{2} \rfloor)) \text{ then } n\ , \\ \text{else } T(f, \lfloor \frac{n}{2} \rfloor) \end{cases}$$

and since $T(f,n) \leq n$, it is clear that there is a feasible λ-term which represents $Smax$.

3.5 Theorem *A type 1 functional $f : \mathbb{N}^k \to \mathbb{N}$ is basic feasible (in the sense of* 3.3*) iff it is polynomial time computable.*

Proof Outline: Every polynomial time computable type 1 function is represented by a feasible λ-term by definition of feasible λ-terms. For the converse, we need a strong normalization theorem (see, e.g. [9]), which states that every λ-term has a β normal form. It is easy to see that a closed type 1 term in normal form has only type 0 variables occurring in it. □

4 Typed While Programs

We start with a simple programming language for computing type 1 functions. All variables $x, y, z \ldots$ are type 0. We allow three kinds of assignment statements:

4.1 $$\begin{cases} x \leftarrow 1 \\ x \leftarrow y + z \\ x \leftarrow y \dot{-} z \end{cases}$$

The only other statement is the while statement:

4.2 While $x \neq 0$ Begin I End

where I is a list of instructions (each instruction is an assignment statement or a while statement). A list of such instructions is called a while-program.

By convention each assignment statement takes one step, and the execution time of a program for a given initialization of variables is the total number of steps. Using standard arguments, including Cobham's theorem [3], we have

4.3 Proposition *A function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ is computable in polynomial time (in the lengths of its inputs) by a while-program iff it is computable in polynomial time on a Turing machine.*

To compute higher type functions, we extend while programs to obtain *typed while-programs* (hereinafter called programs). We introduce variables $X^\tau, Y^\tau, Z^\tau, \ldots$ for each type τ, and we introduce subprograms and more general assignment statements. In addition to the assignment statements 4.1 (which must have only type 0 variables), we allow

4.4 $\quad x \leftarrow X(X_1, \ldots, X_r)$

where x has type 0, X has type $\tau_1 \rightarrow \cdots \rightarrow \tau_r \rightarrow 0$, and X_i has type τ_i, $1 \leq i \leq r$.

Subprograms are like function subprograms in PASCAL, although recursion is disallowed. Each program $\mathcal{P}$ (subprograms are programs) consists of a declaration section and an instruction section. The declaration section consists of three declarations, which have the following forms:

- Input declaration

4.5 $\quad$ Input $Y_1^{\tau_1}, \ldots, Y_k^{\tau_k}$

- Output declaration

4.6 $\quad$ Output x

- Subprogram declarations

$$D_1, \ldots, D_\ell$$

Each subprogram declaration D_i has the form

4.7 $\quad$ Subprogram $V^\sigma : \mathcal{Q}$

where $\mathcal{Q}$ is a program and V^σ is a variable of type σ called the *program name*. V^σ may occur (as X or an X_i) in any assignment statement 4.4 of the instruction section of $\mathcal{P}$, but may not occur in the instruction section or declaration section of any subprogram of $\mathcal{P}$. The type σ of V^σ must coincide with the type of $\mathcal{Q}$, where the type of a program with input declaration (4.5) is $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_k \rightarrow 0$.

The instruction section consists of a list of instructions of the forms 4.1, 4.2 and 4.4 (the body I of a while statement can now include assignment statements 4.4). In general, each variable $X, X_1, \ldots, X_r$ in an assignment statement 4.4 in the instruction section of $\mathcal{P}$ is either a subprogram name in a subprogram declaration of $\mathcal{P}$, an input variable $Y_i^{\tau_i}$ in the declaration 4.5, a local variable, or a parameter (if $\mathcal{P}$ is a subprogram, some of its parameters may be local variables of or inputs to higher level programs). We disallow side effects: any variable y occurring on the left side of any assignment statement in the instruction section of $\mathcal{P}$ cannot be an input or parameter of $\mathcal{P}$ and cannot occur on the left hand side of any assignment statement in a subprogram of $\mathcal{P}$.

If a typed while-program $\mathcal{P}$ of type τ has no global variables and halts on all input functionals, then in a natural way $\mathcal{P}$ *computes* a functional F^τ. This semantics will be specified formally in the full paper. Although all total computable types 1 and 2 functions are computable in this way, not all computable higher type functionals are so computable. One example is the type 3 iterator Φ which takes an operator F of type $(0 \rightarrow 0) \rightarrow (0 \rightarrow 0)$ and number x and composes F with itself x times. That is

$$\Phi(F, x) = F^{(x)}(g_0)$$

where $g_0 = \lambda x.0$. Such higher type iterators are computable (see, e.g. [8]), but an argument similar to that of [12], §8.4, shows that Φ is not while-computable.

We note that Φ can be easily computed by a typed while-program in which subprograms are allowed to call themselves. In fact, it appears that all Kleene computable functionals can be computed by such recursive typed while-programs.

We extend the definition of time complexity to (nonrecursive) typed while-programs by assigning a cost of $|X(X_1, \ldots, X_r)|$ time steps to an assignment statement 4.4. A program $\mathcal{P}$ which computes a functional F with inputs $Y_1, \ldots, Y_k$ and parameters $Z_1, \ldots, Z_s$ is *feasibly length-bounded* if there is a basic feasible functional G such that for all inputs $F_1, \ldots, F_k$ and parameters $H_1, \ldots, H_s$, the time complexity of $\mathcal{P}$ is bounded by $|G(F_1, \ldots, F_k, H_1, \ldots, H_s)|$. A program $\mathcal{P}$ with no subprograms is *feasible* if it is feasibly length-bounded. If $\mathcal{P}$ has subprograms, then $\mathcal{P}$ is *feasible* if it is feasibly length-bounded, and if all of its subprograms are feasible.

4.8 Theorem *A functional F is basic feasible (in the sense of* 3.3*) iff it is computed by some feasible typed while-program.*

Proof: Will be given in full paper. $\square$

Remark If the functional F is type 2, then there is a feasible typed while-program which computes it which has no subprogram declarations. (This corresponds to the normal form for type 2 terms of PV^ω.) From this fact is is easy to see that our type 2 basic feasible functionals essentially coincide with the earlier definitions in Cook [5] and Mehlhorn [16].

Sam Buss (private communication) has given an example of a type 2 functional which is intuitively feasible, in the sense that its run time is bounded by a polynomial in the lengths of the values returned by the input oracles, but does not appear to satisfy Definition 3.3. This is the reason for the word *basic* in basic feasible functional.

However the basic feasible functionals do comprise the smallest class which contains the type 1 poly time functions, is closed under λ-abstraction and application, and has the "Ritchie-Cobham property" for TWP's (i.e. satisfies the analog of Theorem 4.8). This follows from the fact that any class with these properties must contain the $Smax$ functional, which is then sufficient for the definition of a length-bound for a TWP computing $\mathcal{R}$. The definability of $Smax$ follows from the existence of a TWP $\mathcal{P}$ computing the functional $SmaxArg$ which has the property $Smax(f,n) = f(SmaxArg(f,n))$; the running time of $\mathcal{P}$ is polynomial in $|n|$. Note, however, that this result seems to depend on the fact that in the TWP model, the cost of a subprogram call is just the length of the answer.

5 Bounded Typed Loop Programs

Simple bounded loop-programs are similar to while-programs. We allow one more kind of assignment statement:

5.1 $\quad x \leftarrow y\#z.$

The interpretation of this statement is: x is assigned $2^{|y|\cdot|z|}$. Instead of while statements we have bounded loop statements, which have the following form:

5.2 $\quad$ Loop x, y Begin I End

where I is a list of assignment and bounded loop statements. The interpretation of 5.2 is that I is iterated $|x|$ times, with the restriction that no variable is assigned a value with length greater than $|y|$ in any assignment statement inside the loop.

5.3 Proposition *A function $f : \mathbb{N}^k \to N$ is computable by a bounded loop-program iff it is computable in polynomial time on a Turing Machine.*

Bounded typed loop programs (BTLP's) are obtained from bounded loop programs by providing a subprogram structure and assignment instructions (4.4). We also add assignments of the form:

$$x \leftarrow Smax(Y^{0\to 0}, z).$$

(See 3.4.) As expected, we have

5.4 Theorem *A functional F is basic feasible iff it is computed by a BTLP.*

Proof: Will be given in full paper. $\square$

6 Infeasibility Proofs

Suppose F is a feasible type 2 functional taking arguments $X^{0\to 0}$ and y^0 to natural number values (i.e. F has type $(0 \to 0) \to 0 \to 0)$. For each fixed polynomially-bounded $f : \mathbb{N} \to \mathbb{N}$, a BTLP computing $F(f,y)$ will have all variables throughout its computation bounded in length by some polynomials in $|y|$ (depending on f). Hence the value $F(f,y)$ can depend on at most this many values of f. It follows easily that neither the type 2 functional

$$\mathrm{Max}(X,y) = \max_{x \le y} X(z)$$

nor the "zero detector"

6.1 $\quad EB(X,y) = \max_{z \le y} \overline{sg}(X(z))$

is basic feasible (here $\overline{sg}(y)$ is 0 if $y > 0$ and 1 otherwise). We shall see that the situation for type 3 functionals is more subtle.

All computable (and certainly all basic feasible) type 2 functionals $F(X,y)$ are *continuous* in the sense that for each $f : \mathbb{N} \to \mathbb{N}$ and each $a \in \mathbb{N}$ there is a finite set $A \subseteq \mathbb{N}$ such that for all $g : \mathbb{N} \to \mathbb{N}$, if the restrictions of f and g to A agree, then $F(f,a) = F(g,a)$. However, not all type 3 computable functionals are continuous. Consider the functional Φ_1 (adapted from [18], p. 36), defined as follows. For each $m \ge 0$ we define $f_m : \mathbb{N} \to \mathbb{N}$ by

$$f_m(y) = \begin{cases} 0 \text{ if } |y| < m \\ 1 \text{ if } |y| \ge m \end{cases}$$

and $f_\infty(y) \equiv 0$. Then

$$\Phi_1(X) = \begin{cases} 1 \text{ if } \exists m\, X(f_m) \ne X(f_\infty) \\ 0 \text{ otherwise}\,. \end{cases}$$

To see that Φ_1 is discontinuous, let F be the identically zero functional of type $(0 \to 0) \to 0$. Then for each finite subset A of $\{f_0, f_1, \ldots\}$ there is a (continuous) functional G of type $(0 \to 0) \to 0$ so $G(f) = F(f)$ for all $f \in A$, but $\Phi_1(F) \ne \Phi_1(G)$. Nevertheless, Φ_1 is basic feasible. To see this, note that the type 3 functional Ψ_1 defined by

$$\Psi_1(X,y) = \begin{cases} 1 \text{ if } \exists m \le |y|\, X(f_m) \ne X(f_\infty) \\ 0 \text{ otherwise} \end{cases}$$

is easily computed by a BTLP and so it is basic feasible. If we set $g(y) = \Psi_1(X,y)$, then $g = f_\infty$ if

$\Phi_1(X) = 0$ and otherwise $g = f_m$, where m is the least such that $X(f_m) \neq X(f_\infty)$. Thus,

$$\Phi_1(X) = \begin{cases} 1 \text{ if } X(g) \neq X(f_\infty) \\ 0 \text{ otherwise ,} \end{cases}$$

so Φ_1 is basic feasible.

Now for each $m \geq 0$ define g_m by

$$g_m(y) = \begin{cases} 0 \text{ if } y < m \\ 1 \text{ if } y \geq m \end{cases}$$

and

$$\Phi_2(X) = \begin{cases} 1 \text{ if } \exists m\, X(g_m) \neq X(f_\infty) \\ 0 \text{ otherwise .} \end{cases}$$

Then Φ_2 is discontinuous. Is Φ_2 basic feasible? By analogy with the previous argument, we might define

$$\Psi_2(X, y) = \begin{cases} 1 \text{ if } \exists m \leq y\, X(f_m) \neq X(f_\infty) \\ 0 \text{ otherwise .} \end{cases}$$

But now Ψ_2 appears not to be basic feasible (although it is certainly while-computable).

Thus we might conjecture that Φ_2 is not basic feasible. But the simple approach of showing that Φ_2 requires an exponential amount of information about its inputs (as is the case for infeasible type 2 functionals) does not work. Φ_1 is basic feasible, but apparently depends on an infinite amount of information about its inputs. We take a different approach.

6.2 Theorem *Φ_2 is not basic feasible.*

Proof: We show that the infeasible functional EB (given in 6.1) is feasibly reducible to Φ_2. To do this, we define a basic feasible functional $F(X^{0\to 0}, Y^{0\to 0}, z^0)$ with the properties

$$\begin{aligned} F(f, f_\infty, z) &= 0 \text{ for all } f, z, \text{ and} \\ F(f, g_m, z) &= \begin{cases} 1 \text{ if } 0 < m \leq z \text{ and } f(m) = 0 \\ 0 \text{ otherwise .} \end{cases} \end{aligned}$$

The BTLP to compute $F(X, Y, z)$ outputs 0 if $Y(0) \neq 0$ or $Y(z) \neq 1$ and if $Y(0) = 0$ and $Y(z) = 1$ then it uses binary search to find a number $m \leq z$ such that $Y(m-1) = 0$ and $Y(m) \neq 0$, and outputs $\overline{sg}(Y(m))$.

Notice that $EB(X, z) = 1$ iff either $X(0) = 0$ or $\Phi_2(\lambda Y.F(X, Y, z)) = 1$. Since EB is not basic feasible, Φ_2 is not basic feasible. □

References

[1] E. Bishop and D. Bridges. *Constructive Analysis.* Springer-Verlag. 1985.

[2] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. 1986. *Sructure in Complexity Theory*, Springer-Verlag Lecture Notes in Computer Science No. 223, pp. 77-103.

[3] Alan Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel ed., *Proc. of the 1964 Interational Congress for Logic, Methodology, and the Philosophy of Science*, pp. 24-30. North Holland, Amsterdam, 1964.

[4] Robert Constable. Type two computational complexity, *Proc. 5th STOC* (1973) 108-121.

[5] Stephen Cook. The complexity of theorem-proving procedures, *Proc. 3rd STOC* (1971) 151-158.

[6] Stephen Cook and Alasdair Urquhart. Functional interpretations of feasibly constructive arithmetic, *Proc. 21st STOC* (1989) and Technical Report 210/88, University of Toronto, 1988.

[7] K. Ko and H. Friedman. Computational Complexity of Real Functions. TCS 20, 1982, pp. 323-352.

[8] R.O. Gandy. Computable functionals of finite type I. In J. Crossley ed., *Sets, Models, and Recursion Theory* (Proc. of the Summer School in Mathematical Logic and Logic Colloquium, Leicester, England, 1965). pp. 202-242. North Holland, Amsterdam, 1967.

[9] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[10] H.J. Hoover. *Feasibly Constructive Analysis.* Technical Report 206/87, Universtity of Toronto, 1987.

[11] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random number generation for ANY one-way function. STOC '89.

[12] S.C. Kleene. Recursive functionals and quantifiers of finite types I, *Trans. Amer. Math. Soc.* **91** (1959) 1-52.

[13] S.C. Kleene. Recursive functionals and quantifiers of finite types II, *Trans. Amer. Math. Soc.* **108** (1963) 106-142.

[14] S.C. Kleene. Turing-machine computable functionals of finite types I, In P. Suppes ed., *Proc. of the 1960 Congress for Logic, Methodology, and the Philosophy of Science.* pp. 38-45, 1960.

[15] S.C. Kleene. Turing-machine computable functionals of finite types II, *Proc. London Math. Soc.* **12** (1962) 245-258.

[16] Kurt Mehlhorn. Polynomial and abstract subrecursive classes, *JCSS* **12** (1976) 147-178.

[17] A. Nerode, J.B. Remmel, and A. Scedrov. Polynomially graded logic. Manuscript, 1988.

[18] D. Normann. Recursion on the Countable Functionals. Springer-Verlag Lecture Notes in Mathematics, No. 811.

[19] C. Rackoff. Private communication, 1987.

[20] R.W. Ritchie. Classes of predictably computable functions, *Trans. Amer. Math. Soc.* **106** (1963) 139-173.

[21] A.S. Troelstra. Metamathematical investigations of intuitionistic arithmetic and analysis. Springer-Verlag Lecture Notes in Mathematics, No. 344.