Characterizations of the Basic Feasible Functionals of Finite Type *

Stephen A. Cook and Bruce M. Kapron

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 [7] 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 [7] 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 [18] 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 [7]. On the other hand Kleene [14] and [15] introduced and studied the *recursive* functionals of all finite types. Both Kleene [16], [17] and Gandy [9] subsequently gave machine or programming style characterizations of these functionals, but the infinitary nature of these

*A much abbreviated version of this paper appeared as [6] 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 [14], §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 [22] and Cobham [3] that the feasible functionals of [7] 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 $f : \mathbb{N}^k \rightarrow \mathbb{N}$ 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 [11]). 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 [8]. On the other hand, the functional which takes feasible real contractors to their fixed points appears to be feasible ([11], Thm 3.6).

Impagliazzo, Levin and Luby [12] 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 [21], 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 [7].

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 treatment of the typed λ-calculus here follows that in [7], which in turn is patterned after [21].

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 $\quad \tau = \tau_1 \to \tau_2 \to \cdots \to \tau_k \to 0$

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 level* of a type is defined inductively: The level of type 0 is 0, and the level of the type τ in 3.1 is 1 + the maximum of the levels of $\tau_1, \ldots, \tau_k$.

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 length bounded by a fixed polynomial. 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 $0 \to 0$ 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 [23]) 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 labeled with run time bounds (see also [19]). 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 [7], 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 $X^\sigma, Y^\sigma, Z^\sigma, \dots$ 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 $S(T_1, \dots, T_k)$ for $(\dots((ST_1)T_2)\dots T_k)$, and write $\lambda X_1 \dots X_k.T$ or $\lambda X.T$ for $(\lambda X_1.(\lambda X_2.(\dots(\lambda X_k.T)\dots)))$.

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 below. Note that if Z and W are fixed polynomial time computable functions, then $\lambda xy.\mathcal{R}(y, Z, W, x)$ is polynomial time computable. Thus $\mathcal{R}$ preserves polynomial time computability.

In a natural way each closed (i.e. having no free variables) feasible λ-term T of type τ represents a functional in $Fn(\tau)$. This correspondence is demonstrated in the standard way, by showing that a mapping of variables to functionals with corresponding type can be extended to a mapping of terms to functionals with corresponding type.

3.2 Definition A *assignment* is a mapping φ taking variables to functionals with corresponding type. Suppose φ is an assignment and T a term of PV^ω. $\mathcal{V}_\varphi(T)$, *the value of* T *with respect to* φ, is defined by induction on T as follows.

When T is a variable, $\mathcal{V}_\varphi(T)$ is just $\varphi(T)$. If T is a type 1 function symbol, $\mathcal{V}_\varphi(T)$ is the function which is the standard interpretation of T. The value of $\mathcal{R}$ is given by the following equation:

3.3 $$\mathcal{V}_\varphi(\mathcal{R})(y, Z, W, x) = \begin{cases} \text{if } x = 0 \text{ then } y \\ \text{else if } |t| \le |W(x)| \text{ then } t\ , \\ \text{else } W(x) \end{cases}$$

where
$$t = Z(x, \mathcal{V}_\varphi(\mathcal{R})(y, Z, W, \lfloor \tfrac{1}{2}x \rfloor)).$$
Suppose that $\tau = \tau_1 \to \ldots \to \tau_k \to 0$. When T has the form $\lambda X^\sigma.S^\tau$,
$$\mathcal{V}_\varphi(T)(F^\sigma, F_1^{\tau_1}, \ldots, F_k^{\tau_k}) = \mathcal{V}_{\varphi'}(S)(F_1^{\tau_1}, \ldots, F_k^{\tau_k}),$$
where $\varphi'(X^\sigma) = F^\sigma$, but φ' is otherwise identical to φ. When T has the form $S^{\sigma\to\tau}R^\sigma$,
$$\mathcal{V}_\varphi(T)(F_1^{\tau_1}, \ldots, F_k^{\tau_k}) = \mathcal{V}_\varphi(S)(\mathcal{V}_\varphi(R), F_1^{\tau_1}, \ldots, F_k^{\tau_k}). \quad \square$$

It is not hard to show that if T, S are terms such that T is a β or η redex and S is its contractum, then for all φ, $\mathcal{V}_\varphi(T) = \mathcal{V}_\varphi(S)$.

3.4 Definition A functional F is *defined* by a PV^ω term T relative to an assignment φ if $F = \mathcal{V}_\varphi(T)$. A functional is *basic feasible* if some closed PV^ω term defines it.

Note that if T is closed, then $\mathcal{V}_\varphi(T)$ does not depend on φ. So the basic feasible functionals are defined independently of any assignment.

We now give some examples of basic feasible functions. The functional $Apply$ 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

3.5 $$Smax(f, n) = \max_{k \subseteq n} f(k),$$

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
$$T(f, n) = \begin{cases} \text{if } n = 0 \text{ then } 0 \\ \text{else if } f(n) > f(T(f, \lfloor \frac{n}{2} \rfloor)) \text{ then } n\ , \\ \text{else } T(f, \lfloor \frac{n}{2} \rfloor) \end{cases}$$
and since $T(f, n) \le n$, it is clear that there is a feasible λ-term which represents $Smax$.

3.6 Theorem *A type 1 functional* $f : \mathbb{N}^k \to \mathbb{N}$ *is basic feasible (in the sense of* 3.4*) 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. [10] or [21]), 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. The result now follows from the fact that $\mathcal{R}$ preserves polynomial time computability. □

4 Typed While-Programs

We start with a simple programming language for computing type 1 functions, similar to the language introduced in the textbook [13] for computing recursive functions. All variables $x, y, z \ldots$ are type 0. We allow three kinds of assignment statements:

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

Here $y \dot{-} z$ denotes *cutoff subtraction*, namely, $y \dot{-} z$ is 0 if $z \geq y$, and $y - z$ otherwise. The only other statement is the while statement:

4.2 $\mathtt{While}\, x \neq 0\, \mathtt{Do\ Begin}\ I\ \mathtt{End}$

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.

4.3 Proposition *A function* $f : \mathbb{N}^k \to \mathbb{N}$ *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.*

Proof Outline: Each assignment statement at most doubles the value of the largest variable. From this remark it should be clear that a polynomial time bounded while-program can be simulated by a polynomial time bounded Turing machine.

To prove the converse, note that the conditional statement

$$\texttt{If } x < y \texttt{ Then } I \texttt{ Else } I'$$

can be efficiently implemented by while programs using $\dot{-}$ (cutoff subtraction) and standard techniques (see [13]). We argue below that the function $\texttt{Bit}(i, x)$ which gives the ith low-order bit in the binary notation for x is computable in polynomial time by a while-program. From this, it is clear that while-programs can efficiently simulate Turing machines, using standard string processing techniques.

The function $\texttt{Bit}(i, x)$ satisfies the identity

$$x = \sum_{i<|x|} \texttt{Bit}(i, x) \cdot 2^i.$$

Note that 2^i can by computed by a while-program in time $O(i)$ by executing the assignment $x \leftarrow x + x$ exactly i times, starting with $x = 1$.

An outline of a program computing $\texttt{Bit}(i, x)$ in time $O(|x|^2)$ is given in figure 4a □

Remark The preceding result remains valid when we charge $|x|$ steps for an assignment statement 4.1.

To compute higher type functions, we extend while-programs to obtain *typed while-programs*. We introduce new variables $X^\tau, Y^\tau, Z^\tau, \ldots$ for each type τ. We also introduce subprograms and more general assignment statements. In addition to the assignment statements 4.1, we allow

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

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

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

- Output declaration

4.6 $\texttt{Output } x$

```
Input i, x
Output B
 Begin
    y ← x
    B ← 0
    While y ≠ 0 Do
    % Loop invariant: B = 0 and Bit(i, x) = Bit(i, y)
     Begin
        j ← largest k such that 2^k ≤ y  %Bit(j, y) = 1
        If j = i Then
         Begin
            B ← 1
            y ← 0     %exit
         End
        Else y ← y ∸ 2^j
     End
 End
```

Figure 4a: Program computing Bit (statemtents preceded by '%' are comments)

- Subprogram declarations

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

  where $l \geq 0$.

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*. Distinct subprograms must have distinct names. 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 $\mathcal{P}$, and may not be one of the input variables Y_i in 4.5. 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 $\tau_1 \to \tau_2 \to \cdots \to \tau_k \to 0$. Note that it is possible for a program to have no subprogram declarations.

Variables in a typed while-program are partitioned into two sorts: *writable* variables and *read-only* variables. All writable variables are type 0. Only writable variables may appear on the left hand side of an assignment statement 4.1 or 4.4, or as the control variable in a while statement 4.2. All program names in subprogram declarations 4.7 are read-only variables. Variables occurring in an input declaration 4.5 are distinct, must be read-only, and are called *input variables*. The variable occurring in an output declaration 4.6 must be writable, and is called the *output variable*. A writable variable which occurs in the instruction section of a program may not occur in any subprogram declaration of the program. Note that with these restrictions, writable variables are "local", and subprogram calls have no side effects.

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 in an assignment statement 4.1 or 4.4 in the instruction section of $\mathcal{P}$ is either a writable variable, a subprogram name in a subprogram declaration of $\mathcal{P}$, an input variable $Y_i^{\tau_i}$ in the declaration 4.5, or some other read-only variable, called a *parameter* (if $\mathcal{P}$ is a subprogram, some of its parameters may be inputs to higher level programs).

In order to give the meaning of a program in full generality, we need the concept of a *partial functional*. $\mathrm{Pt}(\tau)$, the class of *partial functionals of type* τ, is defined by induction on τ:

- $\mathrm{Pt}(0) = \mathbb{N}$
- $\mathrm{Pt}(\tau_1 \to \ldots \to \tau_k \to 0) = \{F | F : X \to \mathbb{N}, X \subseteq \mathrm{Fn}(\tau_1) \times \ldots \times \mathrm{Fn}(\tau_k)\}$

4.8 Remark We assume a fixed correspondence between read-only variables and variables of PV^ω. Thus we can view an assignment φ as a mapping from read-only variables to total functionals with corresponding type.

For any list I of instructions, any assignment φ, and any sequence $\langle y_1, \ldots, y_m \rangle$ of writable variables containing the writable variables of I, there is a partial mapping $\mathcal{M}_\varphi(I) : \mathbb{N}^m \to \mathbb{N}^m$, the *meaning of* I *with respect to* φ.

Intuitively, the ith ($i \leq m$) component of $\mathcal{M}_\varphi(I)(n_1, \ldots, n_m)$ is the contents of writable variable y_i after the execution of I, assuming that writable variable y_j contains n_j, $1 \leq j \leq m$, before the execution and that all read-only variables have been initialized according to φ. A formal definition is given below.

Consider a program $\mathcal{P}$ of type τ with instruction section I, such that I's writable variables are from $\langle y_1, \ldots, y_m\rangle$, and $\mathcal{P}$'s output variable is y_j, for some $j \leq m$. Suppose also that $\mathcal{P}$ has inputs $Y_1^{\tau_1}, \ldots, Y_k^{\tau_k}$. For an assignment φ, we define $\mathcal{F}_\varphi(\mathcal{P}) \in \mathrm{Pt}(\tau)$, the *functional computed by $\mathcal{P}$ with respect to φ*, inductively. When $\mathcal{P}$ does not have any subprograms,

$$\mathcal{F}_\varphi(\mathcal{P})(F_1, \ldots, F_k) = \pi_i^m(\mathcal{M}_{\varphi'}(I)(0, \ldots, 0))$$

where π_i^m denotes the ith m-ary projection function $\pi_i^m(n_1, \ldots, n_m) = n_i$, and

$$\varphi'(X) = \begin{cases} F_i & \text{if } X \text{ is } Y_i^{\tau_i},\ 1 \leq i \leq k, \\ \varphi(X) & \text{otherwise,} \end{cases} \tag{4.8A}$$

If $\mathcal{P}$ has subprograms $\mathcal{P}_1, \ldots, \mathcal{P}_l$, with names $V_1, \ldots, V_l$ we inductively set $G_i = \mathcal{F}_\psi(\mathcal{P}_i)$, where

$$\psi(X) = \begin{cases} F_i & \text{if } X \text{ is } Y_i^{\tau_i},\ 1 \leq i \leq k, \\ \varphi(X) & \text{otherwise.} \end{cases} \tag{4.8B}$$

Then, assuming that each G_i is total (that is, $G_i \in \mathrm{Fn}(\sigma_i)$ for appropriate σ_i),

$$\mathcal{F}_\varphi(\mathcal{P})(F_1, \ldots, F_k) = \pi_i^m(\mathcal{M}_{\varphi'}(I)(0, \ldots, 0))$$

where

$$\varphi'(X) = \begin{cases} F_i & \text{if } X \text{ is } Y_i^{\tau_i},\ 1 \leq i \leq k, \\ G_i & \text{if } X \text{ is } V_i^{\sigma_i},\ 1 \leq i \leq l, \\ \varphi(X) & \text{otherwise.} \end{cases} \tag{4.8C}$$

If some G_i is not total, then $\mathcal{F}_\varphi(\mathcal{P})(F_1, \ldots, F_k)$ is undefined.

Note that if $\mathcal{P}$ is parameter-free then $\mathcal{F}_\varphi(\mathcal{P})$ does not depend on φ. Hence we can define the unique functional $\mathcal{F}(\mathcal{P})$ which is identical to $\mathcal{F}_\varphi(\mathcal{P})$ for any assignment φ.

It remains to define $\mathcal{M}_\varphi(I)$. In general, if I consists of a single assignment to x_i, then $\mathcal{M}_\varphi(I)(n_1, \ldots, n_m)$ will have the form

$$(n_1, \ldots, n_{i-1}, M, n_{i+1}, \ldots, n_m),$$

for some M. We give the value of M for each type of instruction. For $x_i \leftarrow 1$, $M = 1$; for $x_i \leftarrow y + z$, $M = \varphi'(y) + \varphi'(z)$; for $x_i \leftarrow y \dot{-} z$, $M = \varphi'(y) \dot{-} \varphi'(z)$; and for $x_i \leftarrow Y(Y_1, \ldots, Y_k)$,

$$M = \varphi'(Y)(\varphi'(Y_1), \ldots, \varphi'(Y_k)),$$

where φ' is an extension of φ to writable variables:

$$\varphi'(X) = \begin{cases} n_i & \text{if } X \text{ is } x_i,\ 1 \le i \le m, \\ \varphi(X) & \text{otherwise.} \end{cases}$$

Now suppose that we have defined $\mathcal{M}_\varphi(I')$ and $\mathcal{M}_\varphi(I'')$. If I consists of I' followed by I'',

$$\mathcal{M}_\varphi(I)(n_1, \ldots, n_m) = \mathcal{M}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \ldots, n_m)).$$

Finally, when I has the form `While` $x_i \neq 0$ `Do Begin` I' `End`,

$$\mathcal{M}_\varphi(I)(n_1, \ldots, n_m) = \mathcal{M}_\varphi(I')^{(k)}(n_1, \ldots, n_m),$$

where

$$k = \mu j(\pi_i^m(\mathcal{M}_\varphi(I')^{(j)}(n_1, \ldots, n_m) = 0)).$$

Here, $f^{(k)}$ denotes the k-fold composition of f with itself, and if f is a partial type 1 function with domain a subset of $\mathbb{N}^{k+1}$, $\mu n(f(m, n) = 0)$ denotes the smallest n such that for all $l \le n$, $\langle m, l \rangle$ is in the domain of f, and $f(m, n) = 0$, if such an n exists. Hence $\mathcal{M}_\varphi(I)$ is obtained by the iteration of $\mathcal{M}_\varphi(I')$ the least number of times needed to set the loop control variable to 0. It is clear in this case that $\mathcal{M}_\varphi(I)$ might not be total.

4.9 Definition A partial functional F is *while-computable* if $F = \mathcal{F}(\mathcal{P})$ for some typed while-program $\mathcal{P}$ with no parameters.

Although all total computable types 1 and 2 functions are computable by while-programs of the sort we have been considering, not all computable higher type functionals are so computable. One example is the type 3 iterator Φ which takes an operator F of type $(0 \to 0) \to (0 \to 0)$ 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. [9]), but an argument similar to that of [14], §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. However, it is difficult to define run times for such recursive programs.

We extend the definition of time complexity to (nonrecursive) typed while-programs by defining, for any list I of instructions, any sequence $\langle x_1, \ldots, x_m \rangle$ of writable variables containing the writable variables of I and any assignment φ, a mapping $\mathcal{C}_\varphi(I) : \mathbb{N}^m \to \mathbb{N}$, which is the *cost of* I *with respect to* φ.

Now suppose that $\mathcal{P}$ is a program with instruction section I. Once we have defined $\mathcal{C}_\varphi(I)$, we can then define $\mathcal{T}_\varphi(\mathcal{P})$, *the time complexity of* $\mathcal{P}$ *with respect to* φ, in a way similar to that used to define $\mathcal{F}_\varphi(\mathcal{P})$ from $\mathcal{M}_\varphi(I)$.

Proceeding formally, we use the notation given before 4.8A in the definition of $\mathcal{F}_\varphi(\mathcal{P})$. When $\mathcal{P}$ does not have any subprograms, then

$$\mathcal{T}_\varphi(\mathcal{P})(F_1, \ldots, F_k) = \mathcal{C}_{\varphi'}(I)(0, \ldots, 0),$$

where φ' is defined by 4.8A. If $\mathcal{P}$ has subprograms $\mathcal{P}_1, \ldots, \mathcal{P}_l$ with names $V_1, \ldots, V_l$, we set $G_i = \mathcal{F}_\psi(\mathcal{P}_i)$, where ψ is defined by 4.8B. Then, assuming that each G_i is total,

$$\mathcal{T}_\varphi(\mathcal{P})(F_1, \ldots, F_k) = \mathcal{C}_{\varphi'}(I)(0, \ldots, 0),$$

where φ' is defined by 4.8C. If some G_i is not total, then $\mathcal{T}_\varphi(\mathcal{P})(F_1, \ldots, F_k)$ is undefined.

Notice that no time is charged for the execution of any subprogram $\mathcal{P}_i$ except in the execution of an assignment 4.4, where time $|x|$ is charged (whether or not X is the name of a subprogram.)

We now give a formal definition of $\mathcal{C}_\varphi(I)$. When I consists of an assignment instruction 4.1 or 4.4 into writable variable x_i, then

$$\mathcal{C}_\varphi(I)(n_1, \ldots, n_m) = \max(1, |\pi_i^m(\mathcal{M}_\varphi(I)(n_1, \ldots, n_m))|).$$

In other words, we charge the length of the result, assuming that it is non-zero.

Now suppose that we have defined $\mathcal{C}_\varphi(I')$ and $\mathcal{C}_\varphi(I'')$. Suppose that I consists of I' followed by I''. Then

$$\mathcal{C}_\varphi(I)(n_1, \ldots, n_m) = \mathcal{C}_\varphi(I')(n_1, \ldots, n_m) + \mathcal{C}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \ldots, n_m)).$$

When I consists of a while statement with body I' and loop control variable x_i, we have

$$\mathcal{C}_\varphi(I)(n_1, \ldots, n_m) = \sum_{l<k} \mathcal{C}_\varphi(I')(\mathcal{M}_\varphi(I')^{(l)}(n_1, \ldots, n_m))$$

where

$$k = \mu j(\pi_i^m(\mathcal{M}_\varphi(I')^{(j)}(n_1, \ldots, n_m)) = 0).$$

That is, the cost of a while statement is the sum of the costs for each iteration of the I'.

4.10 Definition Suppose that $\mathcal{P}$ is a program, and that T is a PV^ω term of the same type as $\mathcal{P}$, having free variables coinciding with the parameters of $\mathcal{P}$, such that for all assignments φ,

$$\mathcal{T}_\varphi(\mathcal{P})(F_1, \ldots, F_k) \leq |\mathcal{V}_\varphi(T)(F_1, \ldots, F_k)|,$$

for all $F_1, \ldots, F_k$ with appropriate type. We then say that T *defines a length-bound for* $\mathcal{P}$. $\mathcal{P}$ is *feasibly length-bounded* if some PV^ω term T defines a length-bound for $\mathcal{P}$. $\mathcal{P}$ is *feasible* if it and all of its hereditary subprograms are feasibly length-bounded.

4.11 Theorem ("Ritchie-Cobham theorem for typed while-programs") *A functional* F *is basic feasible (in the sense of* 3.4*) iff it is computed by some feasible typed while-program.*

4.12 Lemma *For any* PV^ω *term* T*, there is a feasible typed while-program* $\mathcal{P}_T$ *having parameters identical to the free variables of* T *such that for any assignment* φ*,* $\mathcal{V}_\varphi(T) = \mathcal{F}_\varphi(\mathcal{P}_T)$*.*

Proof: Proceed by induction on T. In the base case, T is either a single variable, or a function symbol. In the former case, if T is X^τ, we have the following program:

```
Input Y_1^{τ_1}, ..., Y_k^{τ_k}
Output x
 Begin
    x ← X(Y_1, ..., Y_k)
 End
```

For type 1 function symbols, the result follows from 3.6 and the remark following 4.3, and the fact that there are basic feasible functions Pad and $\#$ such that $|Pad(n, m)| = |n| + |m|$ and $|n\#m| = |n||m| + 1$. This leaves us with the case $T = \mathcal{R}$. We give an outline for a program in figure 4b. The reader may verify that there is correct and feasible program based on this outline.

For the inductive step, suppose that for the PV^ω terms $R^{\sigma\to\tau}$, S^σ, we have appropriate programs $\mathcal{P}_R$, $\mathcal{P}_S$, with length-bounds defined by the terms T_R and T_S.

- $T = \lambda X.S$. Suppose that $Y_1, \ldots, Y_k$ are the input variables of $\mathcal{P}_S$. If X is not a free variable of S, $\mathcal{P}_T$ will have the following input declaration:

$$\texttt{Input}\ Z, Y_1, \ldots, Y_k,$$

```
Input y, Z, W, x
Output z
Subprogram MinVal :    %If |Z(u,v)| ≤ |W(u)| return Z(u,v) else W(u)
    Input u, v
    Output w
     Begin
        w ← Z(u,v)
        diff ← |Z(u,v)| ∸ |W(u)|
        While diff ≠ 0 Do
         Begin
            w ← W(u)
            diff ← 0
         End
     End
 Begin    %Main program
    count ← x
    z ← y
    While count ≠ 0 Do
     Begin
        c ← ⌊count/2⌋
        xseg ← x
        While c ≠ 0 Do    %Shift x right by |count| − 1
         Begin
            xseg ← ⌊xseg/2⌋
            c ← ⌊c/2⌋
         End
        z ← MinVal(xseg, z)
        count ← ⌊count/2⌋
     End
 End
```

Figure 4b: Program computing recursor

where Z is some variable not among the variables of $\mathcal{P}_S$. Otherwise, by the induction hypothesis, X is a parameter of $\mathcal{P}_S$, so $\mathcal{P}_T$ may have the following input declaration:

$$\texttt{Input}\ X, Y_1, \ldots, Y_k.$$

Apart from its input declaration, $\mathcal{P}_T$ is identical to $\mathcal{P}_S$. The term $\lambda X.T_S$ defines a length-bound.

- $T = RS$.

  Input $Y_1^{\tau_1}, \dots, Y_k^{\tau_k}$
  Output z
  Subprogram U :
  $\quad\mathcal{P}_R$
  Subprogram V :
  $\quad\mathcal{P}_S$
  Begin
  $\quad z \leftarrow U(V, Y_1, \dots, Y_k)$
  End

  Here we choose $Y_1, \dots, Y_k, U, V, z$ which are not variables of either $\mathcal{P}_R$ or $\mathcal{P}_S$. The term $T_R S$ defines a length-bound.

□

For the converse of the preceding lemma, we first need to bound the size of $\mathcal{M}_\varphi(I)$. This result will be used when obtaining a term interpreting a while statement.

4.13 Lemma *For any list* I *of instructions, if* $\mathcal{M}_\varphi(I)(n_1, \dots, n_m)$ *is defined, then*

$$\max_{1 \leq i \leq m} |\pi_i^m(\mathcal{M}_\varphi(I)(n_1, \dots, n_m))| \leq \max_{1 \leq i \leq m} |n_i| + \mathcal{C}_\varphi(I)(n_1, \dots, n_m)$$

Proof: We proceed by induction on I. For assignment statements 4.1, the result is immediate (since when $n_j = n_i + n_i$, $|n_j| = |n_i| + 1$). For functional calls, the result follows since the cost of the call is the length of the answer returned. Now suppose that the result holds for I' and I''. When I consists of I' followed by I'',

$$\begin{aligned}
&\max_{1 \leq i \leq m} |\pi_i^m(\mathcal{M}_\varphi(I)(n_1, \dots, n_m))| \\
&\quad = \max_{1 \leq i \leq m} |\pi_i^m(\mathcal{M}_\varphi(I'')(M_\varphi(I')(n_1, \dots, n_m)))| \\
&\quad \leq \max_{1 \leq i \leq m} |\pi_i^m(\mathcal{M}_\varphi(I')(n_1, \dots, n_m))| + \mathcal{C}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \dots, n_m)) \text{ (IH)} \\
&\quad \leq \max_{1 \leq i \leq m} |n_i| + \mathcal{C}_\varphi(I')(n_1, \dots, n_m) + \mathcal{C}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \dots, n_m)) \text{ (IH)} \\
&\quad = \max_{1 \leq i \leq m} |n_i| + \mathcal{C}_\varphi(I)(n_1, \dots, n_m).
\end{aligned}$$

The case where I consists of a while statement is handled by an iteration of the preceding case. □

4.14 Lemma *For any list* I *of instructions there is a* PV^ω *term* T_I *with type* $0 \to 0 \to 0$ *and free variables identical to the read only variables of* I *so that for all* $n_1, \dots, n_m \in \mathbb{N}$*, all assignments* φ*, and all* $t \in \mathbb{N}$ *with* $|t| \geq \mathcal{C}_\varphi(I)(n_1, \dots, n_m)$*,*

$$\Pi_i^m(\mathcal{V}_\varphi(T_I)(t, \langle n_1, \dots, n_m\rangle)) = \pi_i^m(\mathcal{M}_\varphi(I)(n_1, \dots, n_m)),$$

for $1 \leq i \leq m$*. Here,* Π_i^m *denotes the* i*th projection function for coded* m*-tuples, and* $\lambda n_1 \dots n_m \langle n_1, \dots, n_m\rangle$ *is an* m*-ary tupling function (note that these functions are basic feasible by* [7], 3.6).

Proof: We proceed by induction on I. The result is immediate when I consists of a single assignment, since constant functions, addition, and cutoff subtraction are all basic feasible. Now assume that for instruction lists I', I'', we have appropriate terms $T_{I'}$ and $T_{I''}$.

When I consists of I' followed by I'', let T_I be $\lambda t n. T_{I''}(t, T_{I'}(t, n))$. Now if $|t| \geq \mathcal{C}_\varphi(I)(n_1, \dots, n_m)$, then $|t| \geq \mathcal{C}_\varphi(I')(n_1, \dots, n_m)$, so by the induction hypothesis,

$$\Pi_i^m(\mathcal{V}_\varphi(T_{I'})(t, \langle n_1, \dots, n_m\rangle)) = \pi_i^m(\mathcal{M}_\varphi(I')(n_1, \dots, n_m)),$$

for $1 \leq i \leq m$. Also, $|t| \geq \mathcal{C}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \dots, n_m))$, so by the induction hypothesis and the preceding equality,

$$\Pi_i^m(\mathcal{V}_\varphi(T_{I''})(t, \mathcal{V}_\varphi(T_{I'})(t, \langle n_1, \dots, n_m\rangle))) = \pi_i^m(\mathcal{M}_\varphi(I'')(\mathcal{M}_\varphi(I')(n_1, \dots, n_m))),$$

for $1 \leq i \leq m$. Hence, by the definition of $\mathcal{M}_\varphi$,

$$\Pi_i^m(\mathcal{V}_\varphi(I)(t, \langle n_1, \dots, n_m\rangle)) = \pi_i^m(\mathcal{M}_\varphi(I)(n_1, \dots, n_m)),$$

for $1 \leq i \leq m$.

Finally, suppose that I is `While` $x_j \neq 0$ `Do Begin` I' `End`. Let U_I be $\lambda t n l. \mathcal{R}(n, Z, W, l)$, where

$$\begin{aligned} Z &= \lambda y z.(\textbf{if } \Pi_j^m(z) = 0 \textbf{ then } z \textbf{ else } T_{I'}(t, z)), \\ W &= \lambda y. s1^{(m)} 0 \# \mathit{Pad}(x, t). \end{aligned}$$

Here $s1$ denotes the basic feasible function $\lambda x. 2x + 1$. We can now show, by notational induction on l, that

$$\Pi_i^m(\mathcal{V}_\varphi(U_I)(t, \langle n_1, \dots, n_m\rangle, l)) = \pi_i^m(\mathcal{M}_\varphi(I)^{(|l|)}(n_1, \dots, n_m)),$$

for $1 \leq i \leq m$. It suffices to verify at each step that

$$|t| \geq \mathcal{C}_\varphi(I')(\mathcal{M}_\varphi(I')^{(|l|)}(n_1, \ldots, n_m),$$

and this follows from the fact that $|t| \geq \mathcal{C}_\varphi(I)(n_1, \ldots, n_m)$. We also use the preceding lemma to verify that W is a sufficiently large bound for the recursion. We now let T_I be $\lambda tn.U_I(t, n, t)$. □

4.15 Lemma *For any feasible typed while-program $\mathcal{P}$, there is a PV^ω term $T_\mathcal{P}$ having free variables identical to the parameters of $\mathcal{P}$ such that for any assignment φ, $\mathcal{F}_\varphi(\mathcal{P}) = \mathcal{V}_\varphi(T_\mathcal{P})$.*

Proof: Suppose that $\mathcal{P}$ has instruction section I, output variable x_i, inputs $Y_1, \ldots Y_k$, and subprograms $\mathcal{P}_1, \ldots, \mathcal{P}_l$ with names $V_1, \ldots, V_l$. Suppose that S defines a length-bound for $\mathcal{P}$, and assume inductively that we have terms $T_{\mathcal{P}_1}, \ldots T_{\mathcal{P}_l}$ corresponding to the subprograms of $\mathcal{P}$. We then let $T_\mathcal{P}$ be the following term:

$$\lambda Y.P_i^m(T_I[T_{\mathcal{P}_1}/V_1] \ldots [T_{\mathcal{P}_l}/V_l](S(Y), \langle 0, \ldots, 0\rangle)),$$

where T_I is defined as in the preceding lemma, and P_i^m defines Π_i^m, and $[T/V]$ denotes the substitution of term T for variable V. □
The theorem is a direct consequence of 4.12 and 4.15.

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 [18].

We have recently discovered an example (to be presented in a future paper) which is intuitively feasible but does not satisfy definition 3.4. 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 4.11). 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 $|n|$. This result seems to depend on the convention 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 $x \leftarrow y\#z$.

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 $\mathtt{Loop}\, x, y\, \mathtt{Begin}\, I\, \mathtt{End}$

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 (y is called a *length-bound* for the loop).

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

Proof Outline: To prove the "only if" direction, one shows by induction on bounded loop programs that the meaning function (the analog of $\mathcal{M}_\varphi(I)$) is always polytime computable.

To prove the "if" direction, we argue that each feasibly length-bounded while-program can be simulated by a bounded loop program. The idea is that the while statement

$$\mathtt{While}\, x \neq 0\, \mathtt{Do}\, \mathtt{Begin}\, I\, \mathtt{End}$$

is replaced by

```
Loop b1, b2
    Begin
        Loop sg(x), b2
            Begin IEnd
    End
```

Notice that the while statement can be executed at most a number of times t bounded by a polynomial in the lengths of its parameters. Now it is easy to see that the function $Pad(x, y) = x \cdot 2^{|y|}$ is computable by a bounded loop program. Now since $|Pad(x, y)| = |x| + |y|$ and $|x\#y| = |x||y| + 1$, for any polynomial $p(x)$ there is a function f which is computable by a bounded loop program so that for all x, $|f(x)| = p(|x|)$. So the number t is bounded by the length of some value $b1$ which can be computed using the assignment statements 4.1 and 5.1, and the function Pad. It follows from Lemma 4.13 that the length bound $b2$ can also be computed from assignment statements 4.1 and 5.1, and the function Pad. Finally, note that $sg(x) = 1 \dot{-} (1 \dot{-} x)$. $\square$

Note that we do not require a notion of time complexity for bounded loop programs. This is because the number of iterations allowed in a Loop statement is feasibly length-bounded. We must bound the length of assignments in a loop to prevent functions with exponential growth rate (recall the definition of exponetiation by repeated squaring). Note that with while-programs this was not necessary, since the cost of an assignment statement is the length of the answer, and so by feasibly length-bounding the running time of a program, we have already imposed a length-bound on assignments (see Lemma 4.13).

Bounded typed loop programs (BTLP's) are obtained from bounded loop programs by providing assignment instructions (4.4) and a subprogram structure (4.5–4.7). We also add assignments of the form:

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

(see 3.5). Variables are classified as either writable or read-only, with the same restrictions as for typed while-programs.

The approach to semantics is similar to that for typed while-programs. Namely, we define a mapping $\mathcal{M}_\varphi(I, b)$ and use this to define $\mathcal{F}_\varphi(\mathcal{P})$. More formally, for any list I of BTLP instructions, any $b \in \mathbb{N}$, any assignment φ, and any sequence $\langle x_1, \ldots, x_m \rangle$ of writable variables containing the writable variables of I, there is a mapping $\mathcal{M}_\varphi(I, b) : \mathbb{N}^m \to \mathbb{N}^m$, the *meaning of* I *with respect to* φ *with assignments bounded by* b.

Intuitively, the ith ($i \leq m$) component of $\mathcal{M}_\varphi(I, b)(n_1, \ldots, n_m)$ is the contents of writable variable x_i after the execution of I, assuming that writable variable y_j contains n_j, $1 \leq j \leq m$, before the execution, that read-only variables have been initialized according to φ, and that no writable variable is assigned a value n such that $|n| > |b - 1|$ (unless $b = 0$, in which case there is no bound on the size of assignments). b is called an *assignment bound*.

In general, when I consists of an assignment statement to the variable x_i, $\mathcal{M}_\varphi(I, b)(n_1, \ldots, n_m)$ will have the form

$$(n_1, \ldots, n_{i-1}, M \upharpoonright b, n_{i+1}, \ldots, n_m),$$

where

$$M \upharpoonright b = \begin{cases} M & \text{if } b = 0, \\ M & \text{if } |M| \le |b-1|, \\ b-1 & \text{otherwise.} \end{cases}$$

Let φ' be the following extension of φ to writable variables:

$$\varphi'(X) = \begin{cases} n_j & \text{if } X \text{ is } x_j,\ 1 \le j \le m \\ \varphi(X) & \text{otherwise.} \end{cases}$$

When I consists of an assignment 4.1 or a functional call 4.4, M is defined in the same way as for typed while-programs. When I is $x_i \leftarrow y\#z$, $M = 2^{|\varphi'(y)|\cdot|\varphi'(z)|}$ and when I is $x_i \leftarrow Smax(Y, y)$, $M = Smax(\varphi(Y), \varphi'(y))$.

Now suppose that we have defined $\mathcal{M}_\varphi(I')$ and $\mathcal{M}_\varphi(I'')$. Suppose that I consists of I' followed by I''. Then

$$\mathcal{M}_\varphi(I, b)(n_1, \ldots, n_m) = \mathcal{M}_\varphi(I'', b)(\mathcal{M}_\varphi(I', b)(n_1, \ldots, n_m)).$$

Finally, suppose that I consists of a loop $\mathtt{Loop}\ y, z\ \mathtt{Begin}\ I'\ \mathtt{End}$. Then

$$\mathcal{M}_\varphi(I, b)(n_1, \ldots, n_m) = \mathcal{M}_\varphi(I', b')^{(|\varphi'(y)|)}(n_1, \ldots, n_m),$$

where

$$b' = \begin{cases} \min(b, \varphi'(z) + 1) & \text{if } b > 0, \\ \varphi'(z) + 1 & \text{otherwise.} \end{cases}$$

The definition of $\mathcal{F}_\varphi(\mathcal{P})$ for BTLP's $\mathcal{P}$ is very similar to that for typed while-programs $\mathcal{P}$, except each occurrence of $\mathcal{M}_{\varphi'}(I)$ is replaced by $\mathcal{M}_{\varphi'}(I, 0)$ (i.e. $b = 0$). This reflects the fact that assignments are not bounded in BTLP's except when they occur within a bounded loop statement.

Note that $\mathcal{M}_\varphi(I, b)$ will always be total, and so $\mathcal{F}_\varphi(\mathcal{P})$ will be total for any BTLP $\mathcal{P}$. As expected, we have

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

Proof Outline: The argument is similar to that for typed while-programs. In particular, for the "only if" direction, the typed while-program given in the proof of Lemma 4.12 for computing the recursor $\mathcal{R}$, can be converted to a BTLP for $\mathcal{R}$, using the method of converting while statements to bounded loop statements outlined in the proof of Proposition 5.3. However now the functional $Smax$ is required to compute the loop bounds. Precisely, the value $x + Smax(W, x)$ can serve as a loop bound for all loops in the Main program. For the "if" direction, an analog to 4.14 is proved in which the term T_I takes the bound b in $\mathcal{M}_\varphi(I, b)$ as an argument, instead of the time t. □

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

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

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 in this sense. Consider the functional Φ_1 (adapted from [20], 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 $f_\infty(y) \equiv 0$. Let X be a variable of type $(0 \to 0) \to 0$. 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

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

```
Input X, x
Output z
Subprogram g_0 :        %λm.(X(f_m) ≠ X(f_∞))
    Input m
    Output z_0
    Subprogram g_1 :  %λy.f_m(y)
        Input y
        Output z_1
        Begin
            m' ← m
            Loop y, m
            Begin
                m' ← m' ∸ 1
            End
            z_1 ← sg‾(m')
        End
    Begin               %g_0
        d_1 ← X(g_1) ∸ X(f_∞)
        d_2 ← X(f_∞) ∸ X(g_1)
        z_0 ← sg(d_1 + d_2)
    End
Begin                   %main program
    z ← 0
    count ← 0
    Loop 2x + 1, 1      %repeat |x| + 1 times
    Begin
        z ← sg(z + g_0(count))
        count ← count + 1
    End
End
```

Figure 6a: Outline of BTLP computing Ψ_1

is easily computed by a BTLP (see Figure 6a) 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

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

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 $\overline{sg}(X(m))$.

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. □

7 Conclusion

We have presented three characterizations of the basic feasible functionals (BFF's). The first characterization is the definition, and uses the typed λ-calculus with a type 2 recursor representing limited recursion on notation.

The second characterization is in terms of typed while-programs, written in a general Algol style programming language for functionals. The BFF's are those functionals which can be computed by such programs in time length-bounded by a BFF. Thus this characterization is not independent of the first, but does provide a strong closure property for the BFF's.

The third characterization is independent of the first two: A functional is a BFF iff it can be computed by a bounded typed loop program.

Taken together, the three characterizations show that the class of BFF's is stable and robust. Nevertheless, there are examples of intuitively feasible type 2 functionals which are not basic feasible, and the authors are studying ways of extending the definition of basic feasible to capture a wider class of feasible functionals.

Section 6 illustrates a nonintuitive property of a type 3 BFF, which suggests that further study of type 3 functionals is needed.

The examples given in section 2 show that functionals pervade theoretical computer science. The time is ripe to develop their complexity theory.

References

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

[2] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. 1986. *Structure 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 International 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 Bruce Kapron. Characterizations of the basic feasible functionals of finite type, *Proc. 30th FOCS* (1989) 154-159.

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

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

[9] 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.

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

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

[12] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random generation from one-way functions, *Proc. 21st STOC* (1989) 12-24.

[13] A.J. Kfoury, R.N. Moll, and M.A. Arbib. *A Programming Approach to Computability*. Springer-Verlag, 1982.

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

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

[16] 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.

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

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

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

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

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

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

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

Department of Computer Science
University of Toronto
Toronto, Canada   M5S 1A4