Comments On Two Notions of Higher Type Computability.

Stephen Bellantoni
University of Toronto

April 1990, with Preface June 1994

1 Preface

I prepared the text of sections 2-6 several years ago for an oral exam at the Department of Computer Science, University of Toronto; it has never been intended for publication. Indeed the text entirely fails to discuss the important Kleene papers circa 1980-1985 that have direct bearing on the suggested program. Nonetheless, it does provide an introduction to higher type computability and might be useful to students of the subject. Concerning higher type complexity theory, see [C] for references, and see chapter 8 of [B2] for a very brief update on the WQO functional and desiderata for a higher type P. The remainder of this preface discusses Kleene's 1980-85 series, assuming some knowledge of the sections 2-6.

Kleene's later (1985) semantics define a class **K** having monotone partial functions **H** ("hereditarily consistent" functions) as inputs to the computation. Kleene does not discuss continuous functions $\dot{\mathbf{C}}$ in his paper [K85], a fact that is in keeping with his extensional treatment of functionals. In some respects, Kleene's later definitions are presaged by Platek's use of hereditarily monotone but non-continuous functions in his dissertation [P].

The relationship between continuous functionals $\dot{\mathbf{C}}$ and the merely monotone functionals **H** is analogous to the relationship between the recursively countable functionals **C** and the arbitrary total functionals **T**: the contrast is continuity versus non-continuity. The functionals in $\dot{\mathbf{C}}$ are a subset of the functionals in **H** up to type level 3, in exactly the same manner as **C** is a subset of **T** up to type level 3. Again, above level 3 the two classes are disjoint since their domains are disjoint; but the problems of interest arise by type level 3. In any case we could easily define $\dot{\mathbf{H}} \subseteq \mathbf{H}$ as a class of functionals which are intuitively continuous in the sense that they are continuous on continuous functionals:

$$\begin{aligned} \dot{\mathbf{H}}_0 &= \mathbf{H}_0 \\ \dot{\mathbf{H}}_{i+1} &= \{h \in \mathbf{H}_{i+1} : \ h(\sqcup X) = \sqcup h(X) \text{ for every chain } X \text{ from } \dot{\mathbf{H}}_i\} \end{aligned}$$

Thus there is a clear sense in which the Kleene semantics of 1985 can be understood as operating on continuous functionals. This eliminates the need for the first step of the program discussed in section 4. The scope of the simplifications is made apparent by observing that the function t, so central to the earlier program, is no longer relevant. Speaking intuitively, the elements of $\mathbf{H}$ are either continuous (in $\dot{\mathbf{H}}$) or not, depending on the values obtained at each point in any chain X of (continuous) inputs. Thus each continuous element Φ in $\dot{\mathbf{H}}$ comes equipped with a specific modulus of continuity that can be inferred from the values of Φ on finite inputs. This is fundamentally different from $\mathbf{T}$ where the only information available was the value of the functional on total (and therefore non-finite) limit points $\sqcup X$ of certain chains of inputs. For $\mathbf{T}$, the modulus of continuity had to be guessed at, and t was used to specify the various possibilities. We can define a function $h : \dot{\mathbf{H}} \to \dot{\mathbf{C}}$ by $h(\Phi)(\alpha) =!\Phi(h^{-1}(\alpha))$ rather analogously to the definition of t, but h does not seem to be as central as t.

A major discovery in Kleene computability was the non-computability of the fan functional — it is a corollary of a result of Tait in 1955 (attributed by [GH]). Does the fan functional remain non-computable under the 1985 definition of Kleene computability? As commented by Cook in [C], the intuitive reason that these functions are not (1958) Kleene computable is that the input functionals can only be queried at total extensions, and therefore it is impossible to find out anything about the intension of the input or how quickly the input becomes defined. This observation no longer holds for Kleene computation over $\mathbf{H}$: the computation can indeed test the value of an input functional at a non-total point. Nonetheless, it is not obvious how to compute the fan or other similar functionals under the later schemas. The later Kleene computations typically diverge if the queried oracle input diverges; therefore it is not clear how the computation can explore the space of finite functionals in order to find one on which the input converges. Intuitively, such a finite functional needs to be found in order to determine an upper bound on the modulus of continuity.

In Scott computability this problem seems to be avoided by use of the parallel operators described in section 2.3 below and in [Pl]. Using an appropriate coding of finite functionals as integers, these operators allow one to test the input at infinitely many integer points (corresponding to infinitely many different finite functionals) and obtain an answer even if the input yields $\perp$ at some of these points.

A major question facing researchers before Kleene's 1980 series was "what schemas should be added to the 1958 Kleene system in order to compute all of the recursively countable functionals?" Given the correspondence (under t) between the recursively countable functionals and the Scott computable functionals cited by Gandy and Hyland [GH] p 434, and given Kleene's revised definitions, the question can be updated as follows. "What schemas, if any, should be added to the 1985 Kleene system in order to compute the Scott computable functionals?". Again, the questions only can be stated formally if we assume a correpondence of $\mathbf{H}$ with $\dot{\mathbf{C}}$ (via $\dot{\mathbf{H}}$) as just discussed. Furthermore, we must restrict attention to the continuous Kleene-computable functionals. The Kleene system of 1985 can easily compute a discontinuous type 2 functional: simply test the input f at all integer values, only returning a result if $f(i) = 0$ for all $i \in \omega$. The question assumes that the Kleene functions – or at least the continuous ones – are contained in the Scott computable functions.

Fundamentally, the introduction of partialness into Kleene computation does not eliminate the lack of parallel features that comprise the difference between $\mathcal{L}_{\mathrm{PA}+\exists}$ and $\mathcal{L}_{\mathrm{DA}}$. It does seem to me that an index elimination proof as in [KM] — with extra proofs to show that Cook's counterexample does not arise during the translation — can prove a correspondence between $\mathcal{L}_{\mathrm{DA}}$ and the 1958 Kleene computations. Carrying this over to the 1980-85 definitions, I conjecture as follows.

Define $\mathbf{K}(\Theta)$ to be the class of functions $\lambda\phi.F(\Theta,\phi)$ where F is Kleene computable under the 1980 schema and Θ is a fixed list of (higher type) inputs. Let $S \subseteq \dot{\mathbf{C}}$ be the Scott computable functionals. Using the mapping $T \to 0$ and $F \to 1$, let $\Theta = \{\hat{\vee}, \hat{\exists}\}$ where $\hat{\vee}$ is the parallel OR function

$$a \hat{\vee} b \equiv \begin{cases} T & \text{if } a = T \vee b = T \\ \text{else} & \begin{cases} \perp & \text{if } a = \perp \vee b = \perp \\ F & \text{otherwise} \end{cases} \end{cases}$$

and $\hat{\exists}$ is the type 2 parallel existential,

$$\hat{\exists}\dot{f} \equiv \begin{cases} T & \text{if } \dot{f}(i) = T \text{ for some } i \in \omega \\ F & \text{if } \dot{f}(\bot) = F \\ \bot & \text{otherwise.} \end{cases}$$

These functions are Plotkin's functions [Pl] adapted to the domain $\mathbf{H}$ instead of $\dot{\mathbf{C}}$.

Conjectures

(1) If Φ has type at most 3, then $\Phi \in \mathbf{K}(\Theta) \cap \dot{\mathbf{H}} \Leftrightarrow \Phi \in S$.

(2) $\Phi \in \mathbf{K}(\Theta) \cap \dot{\mathbf{H}} \Rightarrow h(\Phi) \in S$, and $\dot{\Phi} \in S \Rightarrow \exists \Phi \in h^{-1}(\dot{\Phi}), \Phi \in K(\Theta)$.

The conjectures could also be stated by incorporating Θ into schemas in the style of Kleene, rather than having them as oracle inputs.

I would like to thank Stephen Cook for invaluable discussions on this subject. I am grateful for the perceptive comments of Dag Normann and Samson Abramsky.

2 Introduction

The common view of computation is that the inputs to the computed function are encoded as integer values and these values are input to the machine; the output is an integer which encodes the returned value. Thus, for example, a function on finite graphs is computed using an encoding of graphs. However, such an approach is inadequate when we want to compute a function on objects which are too complicated to encode as integers. For example, consider the partial functional $E(f)$ which is 1 if $f(i) = 1$ for some integer i, and is undefined otherwise. Here, the input $f : \omega \to \{0, 1\}$ is a total function. E is, intuitively speaking, computable by a program such as "$i = 0$; loop: if (f(i)=1) return 1 else i = i+1". We may go a step further and consider a functional such as $\Phi(F)$,

$$\Phi(F) = \begin{cases} 1 & \text{if there is a total function } f : \omega \to \{0, 1\} \text{ s.t. } F(f) = 0 \\ 0 & \text{otherwise} \end{cases}$$

(Here, F is defined on all total inputs). Whether or not Φ should be considered computable depends in part on whether we know a priori that F has a suitable intension.

For our purposes, the extension of a function is the set of input-output pairs which defines the function. An intension of a function is a (smaller) amount of information which determines the function's extension. Generally there will be many intensions for a single functional, since the same extension can be described in many different ways. Not all functions have intensions, because an intension will usually contain less information than the function it represents.

For example, one way to represent an input function intensionally is to present the Godel number of a machine which computes the function. Indeed, the hereditary recursive operators have been defined [B] this basis. Unfortunately this idea does not capture the notions we are after; it gives an algorithm the power to dissect, bitwise, a Godel number of its input. In other words, an HRO computed value depends not just on the extension of the input function, but also on the particular intension that is presented for the input function. By examining the particular intension, an algorithm might find out a fact about the input's extension which is intuitively not computable. A further problem is that the length of the input Godel number is quite unrelated to the execution time of the input functional, making it difficult to define a feasible class of functionals closed under composition. Finally, it becomes impossible to speak about an input function for which we don't happen to have a Godel number.

Another notion of intensionality for a functional $F(f)$ specifies a set of pairs $\langle x, y\rangle$ such that: whenever the input f agrees with the finite amount of information encoded by x, the output of F is y. For example, x might encode an initial segment of values, and y an integer; then $F(f) = y$ if the initial segment of f matches x. Clearly we must consider a countably infinite number of such pairs, because if there were only a finite number of such pairs then the intension of F could be encoded as an ordinary integer. Then we would have gained nothing over our original Turing machine model, in which inputs are integers. This suggests that the intension of a functional F be a simple function mapping integer x to integer y for each pair $\langle x, y\rangle$ as above. This notion can be formalized, even for higher type functionals such as Φ, as an *associate* function mapping integers to integers; the associate completely determines the functional's extension.

We prefer that a computed function depend only on the extension of its input, not on a particular intension which describes the input. Thus a suitable model of computation must refer only to the extension of the input, for example by using oracle calls. Yet it may be that in some models of computation, it is known *a priori* that the input does in fact have an intension; where in others the input does not neccessarily have a suitable intensional representation. This adds to the usual complications which arise due to the different structure of various computation models. Although in ordinary computation theory there is a consensus that Turing machines capture the computable functionals (Church's thesis), there has been less consensus in higher type computability. In order to discuss these issues further, some definitions are needed.

2.1 Some classes of functionals

The set of types is defined by: 0 is a type; if σ and τ are types then so is $\sigma \to \tau$. Using right associativity, every type can be uniquely written as $\sigma_1 \to \sigma_2 \to \ldots \to \sigma_k \to 0$ for some $\sigma_1 \ldots \sigma_k$. This lets us assume that the output of a higher type functional is an ordinary integer. The level of 0 is 0 and the level of $(\sigma_1 \ldots \sigma_k, 0)$ is 1 more than the maximum level of $\sigma_1 \ldots \sigma_k$. With each type $\sigma \to \tau$ we can associate a set $A_{\sigma\to\tau}$ of functionals with domain A_σ and range A_τ. There are many ways of doing this. One common way, which defines the *hereditarily total functionals* is to set $\mathbf{T}_0 = \omega$ and $\mathbf{T}_{\sigma\to\tau}$ = total functions from $\mathbf{T}_\sigma$ to $\mathbf{T}_\tau$.

Using associates, we can (hereditarily) narrow down the class $\mathbf{T}$ to obtain the *countable* functionals. These are total functions whose value is completely determined by a finite amount of information about the (associate of the) input functional. More formally, a functional $F(f)$ is countable if it has a type 1 associate α^F such that whenever x encodes a sufficiently large finite initial segment of α^f, then $\alpha^F(x) = F(f)+1$; otherwise $\alpha^F(x) = 0$. This applies to all higher types; the associate of a type 1 function is just the function itself. It is most easily visualized when F is type 2 and $\alpha^f = f$; then whenever x codes a sufficiently large initial segment of f, $\alpha^F(x)$ defines $F(f)$. The definition can be generalized to non-pure types, i.e. $F(f_1 \ldots f_k) \in \omega$, by encoding finite initial segments of each $\alpha^{f_1} \ldots \alpha^{f_k}$ into x. We write $\mathbf{C}_\sigma$ for the countable functionals of type σ. We think of countable functionals as being a subclass of the hereditarily total functionals, although formally speaking they are different (above type 3) since countable functionals are only defined on countable inputs. A countable functional is *recursively countable* if it has a recursive associate.

An alternative to the hereditarily total functionals are the *hereditarily consistent* functionals, defined by $\mathbf{H}_0 = \omega \cup \{\perp\}$ and $\mathbf{H}_{\sigma\to\tau}$ = total functions F from $\mathbf{H}_\sigma$ to $\mathbf{H}_\tau$. Although the functionals are formally total, they are understood as partial functions in which $\perp$ indicates undefinedness. Furthermore we require monotonicity: if f' extends f, then $F(f')$ extends $F(f)$. Here "extends" refers to the pointwise ordering on functionals in which $\perp \leq i$

for all $i \in \omega$, and $\alpha \leq \beta \Leftrightarrow \forall a, \alpha(a) \leq \beta(a)$. As with $\mathbf{T}$, the value of a functional in $\mathbf{H}$ is not neccessarily determined by a finite amount of information about its input.

By applying a continuity requirement, we can narrow down the hereditarily consistent functionals to those which are determined by a finite amount of information about their inputs. This yields the class $\dot{\mathbf{C}}$ of continuous functionals. More formally, a *complete partial order* is a partial order with a minimum element, $\perp$, such that every directed subset has a least upper bound. A function $\dot{f} : D \to E$ on cpo's D and E is *continuous* iff it is total and $\dot{f}(\sqcup X) = \sqcup \dot{f}(X)$ for every chain X. (Here $\sqcup$ is the least upper bound) Continuity implies monotonicity. If D and E are cpo's, then $(D \to E) \equiv \{\dot{f} : D \to E,\ \dot{f} \text{ continuous}\}$ is a cpo using the induced pointwise ordering (i.e. $\dot{f} \subseteq \dot{g}$ in the usual sense of $\dot{g}$ being more defined than $\dot{f}$). Let $\dot{\mathbf{C}}_0$ be $\{\perp, 0, 1, 2, 3, \ldots\}$ ordered by: $(\dot{x} < \dot{y}) \iff (\dot{x} = \perp \text{ and } \dot{y} \neq \perp)$. Define $\dot{\mathbf{C}}_{(\sigma\to\tau)}$, the structure of continuous functionals of type $(\sigma \to \tau)$, by $\dot{\mathbf{C}}_{(\sigma\to\tau)} = (\dot{\mathbf{C}}_\sigma \to \dot{\mathbf{C}}_\tau)$.

A continuous functional is *Scott computable* if its restriction to finite inputs is recursively enumerable. A finite functional F is one which is defined by a finite number of pairs $< x, y >$, each x encoding a finite functional $\hat{x}$ and each y being the value of F on all $\dot{f} \supseteq \hat{x}$. Encoding finite functionals as integers lets us use the ordinary type 1 notion of recursive enumerability. Because of the continuity requirement, the value of a continuous function is always determined by its value on finite inputs.

A continuous 'everywhere defined' functional corresponds in a certain natural way to a particular associate of a countable functional. The continuous functional agrees with the countable functional on all total inputs, and becomes defined "as soon as" the associate is defined. Ershov [E] showed an exact relationship. We follow Gandy and Hyland [GH] in the following definition of t and everywhere defined functionals, but generalize slightly to non-pure types. The function t maps ED continuous functionals to the corresponding countable functions. Let $\dot{\psi} \in \dot{\mathbf{C}}_\sigma$ where $\sigma = (\sigma_1 \ldots \sigma_k, 0)$; then $t(\dot{\psi}) \in \mathbf{C}_\sigma$ is defined by induction on the type as follows. (We omit type symbols and use vector notation to simplify things, writing for example "$\underline{\dot{\phi}} \in t^{-1}(\underline{\phi})$" to indicate that $\dot{\phi}_i \in t^{-1}(\phi_i)$, $1 \leq i \leq k$).

$$\text{for } \underline{\phi} \in \mathbf{C}, \quad t(\dot{\psi})(\underline{\phi}) = \min y \in \omega \text{ s.t. } \forall \underline{\dot{\phi}} \in t^{-1}(\underline{\phi}),\ \dot{\psi}(\underline{\dot{\phi}}) = y$$

We write $!\dot{\psi}(t^{-1}(\underline{\phi}))$ for the unique value of $\dot{\psi}$ on the equivalence classes $t^{-1}(\underline{\phi})$, if there is a unique value.

Restating the definition, t maps $\dot{\psi}$ to the function which is $!\dot{\psi}(t^{-1}(\underline{\phi}))$ on $\underline{\phi}$ (undefined if $!\dot{\psi}(t^{-1}(\underline{\phi})) = \perp$ or if $\dot{\psi}$ is not unique on $t^{-1}(\underline{\phi})$.) Now we can define the everywhere defined functionals by $\mathrm{ED}_\sigma = \{\dot{\psi} \in t^{-1}(\psi),\ \psi \in \mathbf{C}_\sigma\}$. Intuitively speaking, the everywhere defined functionals are those which are defined and constant on $t^{-1}(\phi)$, for $\phi \in \mathbf{C}$. We can define a pointwise analogue, the *pointwise invariant* functionals, by

$$\{\dot{\psi} : !\dot{\psi}(t^{-1}(\phi)) \text{ exists, for every } \phi \in \mathbf{C}\}.$$

Given a choice, we would prefer $\dot{\mathbf{C}}$ or $\mathbf{C}$ to $\mathbf{H}$ or $\mathbf{T}$, because continuous (or, countable) computations on $\dot{\mathbf{C}}$ (resp., $\mathbf{C}$) can halt after examining a finite amount of information. This hopefully might make it easier to begin measuring the amount of resources used in a computation. Continuous (or, countable) objects are also simpler than non-continuous ones, because the existence of a modulus of continuity (resp, an associate) implies that the number of functionals is the cardinality of the continuum for all type levels above 0. In contrast, the number of $\mathbf{T}$ or $\mathbf{H}$ functionals increases with each type level.

2.2 Kleene computations

In type 1 computation theory (see, e.g. [So]), the primitive recursive functions are defined as the smallest class of functions containing successor, contant, and projection functions, and closed under composition and primitive recursion. To obtain the (partial) recursive functions we add unbounded search or, equivalently, general recursion. Proceeding in a similar manner for higher type functionals, Kleene [K1] gave some schemas S1-S8 for primitive higher type recursion, and an additional schema S9 for general recursion. The class of functions so defined coincides with ordinary recursion theory for type 1, because the higher type schema are not used in computing a type 1 function. The Kleene schema are indexed by integers, which leads to s-m-n and recursion theorems. An equivalent class can be defined without referring to indices ([P], [KM]) by using combinators and fixed points. To access a higher type input ψ_1 from the list $\underline{\psi}$ of inputs, S8 allows that if Ψ is computable, then so is

$$\Psi'(\underline{\psi}) = \psi_1(\lambda\alpha.\, \Psi(\alpha, \underline{\psi})).$$

Kleene understood his schema in terms of hereditarily total functionals, so that this expression is undefined whenever $\lambda\alpha.\, \Psi(\alpha, \underline{\psi})$ is not total. On the other hand, Kleene computable functionals are not total, so clearly the class of partial Kleene computable functionals is not closed under composition.

In fact, even the result of substituting a total Kleene functional into a Kleene computation is not Kleene computable (see, e.g. [KM]). Roughly speaking, the reason is that the traditional Kleene S8 allows one to test whether or not a computed function is total — if this ability is allowed in a type 1 function then one can contradict the equivalence of type 1 Kleene computation and ordinary recursion theory. The composition $H(y) = F(\lambda z.G(y,z))$ (where F and G are Kleene computable and y is type 0) would serve to reduce the computed type of H to type 1, while possibly still retaining an application of S8 in F. The composition is computable, however, if F is total.

For our purposes it will be convenient to think of S1-S9 in terms of countable functionals rather than hereditarily total functionals. **C** is identical to **T** for type levels 0 and 1, and at type level 2 **C** is a proper subset of **T**. It follows that the following two notions are identical at least up to and including a type 3 computed functional: (`i`) the functional is a Kleene computable functional (interpreted over **T**) restricted to be a function of countable inputs; and (`ii`) the functional is a Kleene computable functional (interpreting the λ appearing in S8 as ranging over **C**, and with inputs in **C**). All the problems we consider arise as early as type level 3, so we are satisfied with just using the countable interpretation for "Kleene computable".

How do the Kleene computable functionals compare to the recursively countable functionals? Every Kleene computation using inputs $\underline{\psi}$ is equivalent to a computation (in the ordinary sense) using as oracle inputs the associates of $\underline{\psi}$ [GH]; so we know that every Kleene computable functional is recursively countable. The converse is false. For example, the type 3 zero detector Φ mentioned earlier was observed by Cook (following a proof of Gandy and Hyland [GH]) to be recursively countable but not Kleene computable. Gandy and Hyland's *fan functional* example, too, shows that there is no Kleene program which computes the *modulus of continuity* of its input. The modulus of a type 2 functional F (on 1), is the least n such that for all $f : \omega \to \{0,1\}$, $F(f)$ is completely determined by the n initial values of f. The modulus tells us something about what possible associates F could have. Philosophically these examples are interesting because they suggest that there is no Kleene program for computing information about the intensions of the input. This is natural since Kleene understood the inputs to be extensionally presented functionals in **T**, not all of which even have associates (i.e. intensions).

2.3 Continuous LCF Computations; and a Conjecture

In a separate and later line of development, Scott [Sc] defined the Logic of Computable Functions based on the lambda calculus, taking the denotation of LCF terms to be elements of a lattice ordered by definedness. This formed the basis of Ershov's definition [E] of cpo's which has been used since. Plotkin [Pl] examined variations on LCF, comparing the denotation of LCF terms in cpos to the operational results of applying reduction rules on the LCF term.

Our interest in Plotkin's paper is the discussion of the three languages $\mathcal{L}_{\mathrm{DA}}$, $\mathcal{L}_{\mathrm{PA}}$ and $\mathcal{L}_{\mathrm{PA}+\exists}$. The first of these consists of terms in Scott's LCF; namely, terms formed from integers, typed variables, addition, zero test, conditional branching, lambda abstraction, application, and the least fixed point operator (Y). The second language, $\mathcal{L}_{\mathrm{PA}}$, adds a parallel OR operation which is true if either of its arguments halts and outputs true; false if they both halt and output false. $\mathcal{L}_{\mathrm{PA}+\exists}$ is $\mathcal{L}_{\mathrm{PA}}$ further augmented by the type 2 parallel $\hat{\exists}$ function, defined by:

$$\hat{\exists}\dot{f} = \begin{cases} 1 & \text{if } \dot{f}(i) = 1 \text{ for some } i \in \omega \\ 0 & \text{if } \dot{f}(\perp) = 0 \\ \perp & \text{otherwise} \end{cases}$$

Notice that by the monotonicity condition, $\dot{f}(\perp) = 0$ implies that $\dot{f}(i) = 0$ for all $i \in \omega$. One explanation for the parallel $\hat{\exists}$ is that each of the subcomputations "$\dot{f}(i) = 1$" for $i \in \omega$ are run in parallel along with "$\dot{f}(\perp) = 0$"; whatever computation halts with output 'true' determines the result.

As mentioned earlier, the Scott computable functionals are those continuous functions whose set of restrictions to finite inputs is recursively enumerable. For example, consider type 1 functions that are defined on a finite number of (finite) inputs; each integer e is the code of some such function $[e]$. Then $F \subseteq \dot{\mathbf{C}}$ of type 2 is Scott computable iff $\lambda x.F([x])$ is partial recursive in the ordinary sense. (See [C3] for a more complete definition).

Plotkin shows that $\hat{\exists}$ is not computable in $\mathcal{L}_{\mathrm{PA}}$ and that the Scott computable functionals are exactly those denoted by $\mathcal{L}_{\mathrm{PA}+\exists}$ terms. Gandy and Hyland note that everywhere defined Scott computable terms are the pre-images, under t, of the recursively countable functionals ([GH] p 434). Thus everywhere defined $\mathcal{L}_{\mathrm{PA}+\exists}$ terms denote, under t, the recursively countable functionals — including such non-Kleene computable functionals as the fan functional. It seems plausible that the lack of parallel facilities in Kleene computation accounts for the inability to Kleene compute the fan functional

[P]. Having noticed this, Cook asked whether the $\mathcal{L}_{\rm DA}$ terms denote, under t, the Kleene computable functionals. This conjecture can be stated in two forms. The weak form is, that the total Kleene computable functionals are exactly the image under t of the everywhere defined $\mathcal{L}_{\rm DA}$ functionals. The stronger form is, that the Kleene computable functionals are exactly the image under t of the pointwise invariant $\mathcal{L}_{\rm DA}$ functionals.

3 The conjecture's negation

The opposite of the weak conjecture would be: every recursively countable functional is the image under t of an everywhere defined $\mathcal{L}_{\rm DA}$ term. At first glance one might think that $t(\hat{\exists})$ would provide a counterexample, because $\hat{\exists}$ is not $\mathcal{L}_{\rm DA}$ definable. However, $\hat{\exists}$ is not everywhere defined. Similarly, Platek [P] comments that the parallel OR is not Kleene computable, because of the deterministic nature of Kleene compuataion; but neigher is it everywhere defined. Interestingly we can show that $\hat{\exists}$ is equivalent (using $\mathcal{L}_{\rm PA}$ reductions) to the type 3 everywhere defined function $\dot{\Phi}$,

$$\dot{\Phi}(\dot{F}) = \begin{cases} \perp & \text{if there exists everywhere defined } \dot{\alpha} \leq 1,\ \dot{F}(\dot{\alpha}) = \perp \\ \text{else} & \begin{cases} 1 & \text{if there exists } \dot{\alpha} \leq 1,\ \dot{F}(\dot{\alpha}) = 0 \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

(Here we have written "$\dot{\alpha} \leq 1$" for "$\forall i \in \omega,\ \dot{\alpha}(i) \downarrow \leq 1$"). The proof follows the method of Plotkin [Pl] for searching among finite functionals. Noticing that $t(\dot{\Phi}) = \Phi$, the fact that Φ is recursively countable, yet $\dot{\Phi}$ is not $\mathcal{L}_{\rm DA}$ computable, strongly suggests that not all recursively countable functionals are the images under t of $\mathcal{L}_{\rm DA}$ terms. But the implication does not follow, because there might be some $\mathcal{L}_{\rm DA}$-denotable term $\dot{\Phi}' \neq \dot{\Phi}$ such that $t(\dot{\Phi}') = \Phi$. We would have to prove the $\mathcal{L}_{\rm PA}$ non-computablity of $\dot{\Phi}$ using only continuity and the value of $\dot{\Phi}$ on everywhere defined inputs. In short, the counter-conjecture is not that every $\mathcal{L}_{{\rm PA}+\exists}$-denotable ED functional is also $\mathcal{L}_{\rm DA}$-denotable; but that every $\mathcal{L}_{{\rm PA}+\exists}$-denotable ED functional is equivalent via t to an $\mathcal{L}_{\rm DA}$ denotable functional.

4 Approaching the Conjecture.

One approach we have tried for proving the conjecture is as follows. First reinterpret the Kleene schema, using $\dot{\mathbf{C}}$ and relaxing the awkwardness of S8 so that an output may be obtained even if the input functional is not total; also, prove composition theorems for these continuous Kleene computations. Second, show the equivalence of continuous Kleene computations and $\mathcal{L}_{\mathrm{DA}}$. Finally, argue that if a Kleene index computes an everywhere defined functional $\dot{f}$ under the continuous semantics, then $t(\dot{f})$ is computed by the same index under the traditional semantics.

The first part of this program can be carried out with a minimum of difficulty. The continuous semantics are substantially different from the traditional semantics in that the output of a continuous computation is determined by the values assigned to nodes in a finite computation tree, where the traditional semantics requires that values be assigned to an infinite number of subcomputation nodes. Also, the composition theorems which fail for the traditional semantics [KM], [K1] can now be carried through, because the continuous version of S8 does not give any information about whether a computed function is total.

The second part of this program can also be carried out. Some detailed syntactic transformations are required in order to code the input in an appropriate way: where Kleene computations have a varying number of pure type inputs, $\mathcal{L}_{\mathrm{DA}}$ inputs consist of a fixed number of variables of any type. Regardless, in one direction we can write an $\mathcal{L}_{\mathrm{DA}}$ interpreter which executes arbitrary continuous Kleene indices. This is a detailed but essentially straightforward proof. In the other direction we can construct continuous Kleene indices for arbitrary $\mathcal{L}_{\mathrm{DA}}$ terms. The difficult part is handling $\mathcal{L}_{\mathrm{DA}}$'s least fixed point operator, Y; for this we can use some ideas from the index-free approach to Kleene computation, e.g. in [KM]. Roughly speaking, in the continuous context the least fixed point can be understood as recursion: in a term "$Y\lambda\alpha\gamma$", we simply begin executing the index e_γ for γ; when α is encountered in γ we perform a recursive execution (S9) of e_γ. The index e_γ is obtained from this self-referential scheme using the second recursion theorem. In summary, the equivalence of $\mathcal{L}_{\mathrm{DA}}$ terms and continuous Kleene computations can be seen as a continuous analog of Platek's [P] result, which gives an index-free combinator equivalent of Kleene's S1-S9 interpreted over hereditarily consistent inputs.

The third part of this program, showing the equivalence of traditional and ED continuous Kleene functionals, fails as stated. It can be proved that whenever a value is obtained on input $\underline{\psi}$ using the traditional semantics, then that same value is obtained for all inputs in $t^{-1}(\underline{\psi})$ using the continuous semantics. And at first the reverse also seems to be true. For, the difference in convergence is primarily that the continuous scheme

$$\dot{\psi}_1(\lambda\dot{\beta}\dot{\Psi}(\dot{\beta},\underline{\dot{\psi}}))$$

might output a defined value even when $\lambda\dot{\beta}\dot{\Psi}(\dot{\beta},\underline{\dot{\psi}})$ is not everywhere defined. Since $t(\,\lambda\dot{\beta}\dot{\Psi}(\dot{\beta},\underline{\dot{\psi}})\,)$ is not total, the corresponding traditional computation diverges. Yet for any particular such $\dot{\Psi}$ we can choose $\dot{\psi}_1' \in t^{-1}(\psi_1)$ such that $\dot{\psi}_1'(\lambda\dot{\beta}\dot{\Psi}(\dot{\beta},\underline{\dot{\psi}}'))=\perp$. If the scheme outputs a unique value on all elements of $t^{-1}(\underline{\psi})$, it must output $\perp$ on all these elements, thus coinciding with the divergence of the traditional computation. Unfortunately, a closer analysis reveals that we might have to apply this argument an infinite number of times, which we cannot. More concretely, Cook showed that it is possible to construct an index such that on each member of $t^{-1}(\underline{\psi})$, the continuous computation outputs a value after recursing to a finite depth, although there is no single finite bound for the recursion depths on all the inputs in $t^{-1}(\underline{\psi})$. Under the traditional semantics the corresponding recursion is infinitely deep, and the computation diverges.[1]

However, all is not lost in this three stage approach. For, the index used as a counter-example above is unusual in its usage of recursion. Specifically, there are an infinite number of different sub-indices which are executed by

[1]Let $\dot{\psi}$ be type 3, let $\dot{e}_i$ be the index which performs an S8 to get $\{\dot{e}_i\}(\dot{\psi}) = \dot{\psi}(\lambda x\{\dot{e}_i'\}(x,\dot{\psi}))$ and such that $\{\dot{e}_i'\}(x,\dot{\psi}) = 0$ if $x \neq \dot{e}_{i+1}$, and (using S9) $\{\dot{e}_i'\}(x,\dot{\psi}) = \{x\}(\dot{\psi})$ otherwise. Here, $\dot{e}_{i+1} > \dot{e}_i$ has the same properties as just described for $\dot{e}_i$. The index $\dot{e}_{i+1}$ is Kleene computable from $\dot{e}_i$, which can be obtained by the recursion theorem. Notice that the functionals $\lambda x, \{\dot{e}_i'\}(x,\dot{\psi})$ are identically zero on an initial prefix $0,1,\ldots,n_i$ whose length n_i is increasing with i. Hence, letting $\psi \in \mathbf{C}$, for every $\dot{\psi} \in t^{-1}(\psi)$ there is an i (the modulus of $\dot{\psi}$) such that $\dot{\psi}(\lambda x, \{\dot{e}_i'\}(x,\dot{\psi})) \neq \perp$, implying that $\{\dot{e}_0\}(\dot{\psi}) \neq \perp$. In fact $!\{\dot{e}_0\}(t^{-1}(\psi))$ can be seen by considering the least i such that ψ is constant, say c, on all functions which are zero on $0\ldots n_i$. There is such an i because $\psi \in \mathbf{C}$ is continuous. Regardless of the modulus of $\dot{\psi}$, $\dot{\psi}(\lambda x, \{\dot{e}_i'\}(x,\dot{\psi})) = c$. Proceeding upward along the chain, we obtain a fixed value for $\{\dot{e}_i\}(\dot{\psi})$ regardless of the modulus of $\dot{\psi}$. In summary, there is an index which computes a constant function independent of the modulus of it's argument, and which contains an infinite descending chain. The corresponding traditional Kleene index cannot converge, because of this infinite chain. However, there still might be some other traditional Kleene index computing the same funtional. N.B.: we already have a separate and much simpler proof that if $\{e\}(\psi)\downarrow$ then $\{\dot{e}\}(\dot{\psi}) = \{e\}(\psi)$ for all $\dot{\psi} \in t^{-1}(\psi)$.

the recursion scheme (S9). While the counterexample does show that some ED indices compute different functions under the continuous and traditional interpretations, it does not clearly show that indices produced by our translation have this problem. In fact, a casual inspection of the translation from $\mathcal{L}_{\mathrm{DA}}$ to continuous Kleene indices seems to indicate only a finite number of different sub-indices which are executed by S9. It seems plausible that no counterexample can be constructed executing only a finite number of different sub-indices. Intuitively speaking, it appears that the counterexample uses recursion in an unusual way, whereas the translation only uses recursion in the ordinary way. If this intuition can be formalized as a statement about the computation graphs of translated $\mathcal{L}_{\mathrm{DA}}$ terms, then a proof of the conjecture might well follow.

5 Desiderata for a Higher Type P

A higher-type analogue for the familiar poly-time computability class should satisfy three conditions: (1) its type 1 functions should be exactly the functions in P; (2) it should be closed under lambda abstraction and application; (3) it should contain all intuitively feasible functionals. Cook [C] describes (3) as requiring at least the following reasonable condition: (4) its type 2 functions should be computable by an oracle Turing machine in time polynomial in the length of the type 0 inputs and the values returned by oracle calls.

One approach to defining a higher type feasible class would be to give a machine characterization, similar to Turing machines, of the Kleene computable functionals, and then restrict the running time of these machines. In fact, Kleene [K2] did give a machine characterization of his functionals, using a Turing machine with an oracle which works as follows: to query $\psi_1(\lambda\beta\Psi(\beta,\underline{\psi}))$, one writes down the Godel number of the computation defining Ψ on the oracle tape, and then enters the oracle query state. The intuition is that the oracle will execute the given machine whenever it needs a value of Ψ. Unfortunately, in using Godel numbers here we run into the same problems as occur in the HRO functionals. For example, let $A(f,x) = f(x)$ be the type 2 "Apply" functional — it is feasible. Then the infeasible function $\lambda n.F(A,n) = \lambda n.2^n$ can be computed by writing down the Godel number of $\lambda x.2^x$ and using the Apply oracle. Of course, we have evaluated the oracle on an infeasible input functional. However, we can compute this even if we restrict the oracle input (the Godel number) to be that of a feasible functional: we can write down the Godel number of $\lambda x.2^n$ and then call the Apply oracle. Since $\lambda x.2^n$ is a constant function, it is feasible. (This example due to Kapron). Finally, it is easy to see that to get a feasible class using Kleene's machines, we have to add a length bound on computed values — otherwise we could compute $\lambda n.2^n$ by repeated squaring.

An alternative approach to feasibility is to say the feasible functionals are those denoted by terms in a suitable language, such as PV^ω [CU]. This language, which is interpreted over $\mathbf{T}$, is based on typed lambda calculus starting with all type 1 polytime functions, together with type 2 bounded recursion on notation. The recursion allowed is, intuitively, ordinary primitive recursion except that instead of defining, say, $f(x)$ in terms of $f(x-1)$, one may only define $f(x)$ in terms of $f(\lfloor x/2 \rfloor)$. Hence the depth of the recursion is $|x|$ instead of $2^{|x|}$; a term with k nested recursions corresponds to a depth of $|x|^k$. Additionally, the value computed during the recursion is never allowed to be larger than a value which can be computed by a pre-determined PV^ω term (i.e. in polynomial time). The PV^ω definable functions can be characterized by a programming language, the bounded typed loop programs [CK] in which each loop executes a bounded number of times and performs assignments whose length is bounded by a previously computed value. The loop bounds can be removed if one bounds the total length of assigned values by a PV^ω-definable function of the inputs.

Interestingly, there is at least one functional which is not PV^ω definable yet is, arguably, feasible [C]. Intuitively, the reason is that PV^ω computable functionals, at each step, can only access the oracle at a point which is polynomially computable from oracle values previously accessed. While a feasible functional may be computable in time polynomial overall in the oracle outputs, the PV^ω term is restricted by having to examine the oracle incrementally, starting at the initial values of the oracle.

A third approach to defining a class of higher type feasible functionals is to bound the running time of the functional's associate. Recall that an associate (e.g. of a type 2 functional) is any function suitably defined on some sufficiently long initial segment of the input. Clearly, then, it is inadequate to simply require that there be a polynomial time associate — an associate can be undefined until the input initial segment is exponentially larger than is really needed to compute the output value. This "padding" of the input can give the associate as much time as it wants, to compute the answer. On the other hand there is a unique *principal* associate which is defined as soon as possible *i.e.* on the shortest possible initial segment of the (type 1) input. This observation led Kapron and Cook to propose that a function might be considered feasible if it has a polytime "honest" associate, where honest means that the padding doesn't occur. Formalizing the notion of an honest associate is not very easy. I think one could define an honest associate to be one which is defined on an initial segment whose length is at most polynomially more than the length of the minimum initial segment, *i.e.* polynomially more than the length of the shortest initial segment for which the principal associate is defined. Intuitively, it is neccessary to allow the honest associate some padding of the input segment. Otherwise the only honest associate would be the principal associate, which might not even be recursive [GH]. The relationship between the functionals which have polytime honest associates and other complexity classes, such as the Kleene computable functionals and the PV^ω computable functionals, is quite open.

Yet another approach to feasibility considerations would try to extend the notion of polynomial time directly, by introducing higher type polynomials. We could attach to each functional F two functionals F^t and F^s having the same type as F. These would be understood to be the time and space complexity of F. Then one might require the computation time of a computed functional to be a polynomial in the time and/or space functions of its inputs. For a call on oracle F at computed input g, one would charge a $F^t(g^t, g^s)$ time units. At type 1, $f^t(x)$ would have to correspond to ordinary Turing machine time on input x, and $f^s(x)$ would be monotone increasing with x. The implications of such an approach remain to be explored.

6 Conclusion

Higher type computation is of interest both intrinsically and for specific reasons such as its usage in Kreisel realizability and Godel's *Dialectica* translation. The objects of study are much richer and more complex than the plain integers of ordinary computation theory. Further, one can imagine applications to such areas as A.I. where one frequently reasons about relations of relations of objects, etc.

On the other hand, there are a profusion of closely intertwined ideas all of which are difficult to visualize concretely at and above type level 4. It seems clear that restricting attention to countable or continuous functionals will be beneficial because of the nice properties these functions enjoy and because continous or countable computations can halt after examining a finite segment of the input. Yet, the models of higher type computation most frequently been used for theorizing computational complexity have been interpreted over the hereditarily total domain. Furthermore, no entirely satisfactory notion of time complexity has been developed. These subjects provide many topics for further research.

References

[B2] Bellantoni, Stephen J.: "Predicative Recursion and Computational Complexity", TR 264/92, Department of Computer Science, University of Toronto, September 1992.

[B] Buss, Samuel R.: The Polynomial Heirarchy and Intuitionistic Bounded Arithmetic. In *Structure in Complexity Theory,* Springer-Verlag Lecture Notes in Computer Science, (1986) #223, p. 77+

[C] Cook, S.: Computability and Complexity of Higher Type Functionals *Proc. MSRI Workshop on Logic from Computer Science*, Y. Moschovakis ed., (1990).

[C2] Cook, S.: Computational Complexit of Higher Type Functions, in *Proc. of the International Congress of Mathematicians, 1990.* (ICM-90).

[CU] Cook, S. and Urquhart, A.: Functional Interpretations of Feasibly Constructive Arithmetic. Technical Report 256/91 (revision of 210/88), Department of Computer Science, University of Toronto 1991 (revised from 1988).

[CK] Cook, S. and Kapron, B.: Characterizations of the Basic Feasible Functionals of Finite Type. *Proc. MSI Workshop on Feasible Mathematics*, Cornell University (1990)

[CK2] Cook, S. and Kapron, B.: A New Characterization of Mehlhorn's Polynomial Time Functionals, Proc. Foundations Of Computer Science, 1991.

[E] Ershov, Y.: Maximal and Everywhere Defined Functionals. *Algebra and Logic* (1972) v.13, p.210+

[GH] Gandy, R. and Hyland, J.: Computable and Recursively Countable Functions of Higher Type. *Logic Colloquium 76* North Holland (1977) p.407+

[KM] Kechris, A. and Moschovakis, Y.: Recursion in Higher Types. *Handbook of Mathematical Logic*, J. Barwise, ed., North-Holland (1977)

[K1] Kleene, S.: Recursive Functionals and Quantifiers of Finite Types I. *Transactions of the American Math Society*, (1959) v.91, p.1+

[K2] Kleene, S.: Turing Machine Computable Functionals of Finite Types I. *Proc. 1960 Congress for Logic, Methodology, and the Philosophy of Science*, P. Suppes, ed., (1960) p.38+

[K3] Kleene, S.: "Unimonitone Funtionals of Finite Types (RFQFTR V)", *Proceedings of Symposia in Pure Mathematics*, v. 42, 1985.

[P] Platek, R.: Foundations of Recursion Theory. *Ph.D. Thesis*, Stanford University, 1966.

[Pl] Plotkin, G.: LCF Considered as a Programming Language. *Theoretical Computer Science*, v. 5, p. 223, 1977.

[Sc] Scott, D.: Outline of a Mathematical Theory of Computation. *Proc. of the 4th Annual Princeton Conference on Information Science and Systems*, (1970) p.169+

[Se1] Seth, Anil.: "There is No Recursive Axiomatization for Feasible Functionals of Type 2", manuscript, Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005 (seth@tifrvax.bitnet)

[So] Soare, Robert.: *Recursively Enumerable Sets and Degrees* Perspectives in Mathematical Logic series, Springer-Verlag (1986)

[Si] Thompson, Simon: "Recursion Theories On The Continuous Functionals", Ph.D. Thesis, New College, Oxford, March 1984.