Computing with functionals - computability theory or computer science?

Dag Normann [*]

March 10, 2004

Abstract

We review some of the history of the computability theory of functionals of higher types, and we will demonstrate how contributions from logic and theoretical computer science have shaped this still active subject.

1 Introduction

The AMS 2000 Mathematics Subject Classification has an entry, 3D65, called *Higher type and set recursion theory.* Important and ingenious work on computability relative to normal functionals, relative to the superjump or in set recursion was carried out under this heading in earlier days. If one thinks that this kind of research is the hard core of the subject, it is fair to say that the subject faded away in the early 90'ies. There is, however a different story to tell.

Some logicians, including the author, was interested in the computability theory of the continuous functionals. Some computer scientists[1] have also taken an interest in higher type computations. Fortunately the two groups have not been totally unaware of each other, but there were surprisingly little contact for some years. Lately, this has changed. One example of this is the survey paper [23] by John Longley. By our operational definition, Longley is a computer scientist, but he is equally based in the traditions of theoretical computer science and in computability theory.

The aim with this paper is not to write the full story of the computability theory of typed functionals, and even less the full story of how higher type objects in various forms have played a role in computer science. Through examples of approaches and results we just want to demonstrate how the subject has been influenced both from mathematical logic and from theoretical computer science, serving the interest of both communities. In our glimpses from the early history

[*]Department of Mathematics, The University of Oslo, P.O. Box 1053, Blindern N-0316 Oslo, Norway. e-mail: dnormann@math.uio.no

[1] A computer scientist will be someone with a research position at a computer science department.

we will focus on some conceptual approaches that was influencial on what is forming the subject today.

It is sometimes claimed that continuity is a sound abstraction of computability. In our view topology is a useful tool, but does not, even at an abstract level, capture the nature of computability. We actually share the view of prominent computer scientists that some kind of sequentiality, modelling deterministic procedures in some abstract way, would be more apropriate than just continuity. In discussing the recent history we focus on the search for the understanding of sequentiality. We will give a brief, but self-contained construction of a promising candidate for the sequential continuous partial functionals of finite types.

We have not aimed to mention all important contributions to the subject under discussion. For instance, Moschovakis and Fefferman gave important technical and conceptual contributions linking generalised computability theory to issues of genuine computation. Our choice has been to guide the reader towards the search for, and understanding of, sequentiality, and the selection of contributions we mention is dictated by this choice.[2]

Convention

We will assume that the reader at least has a vague idea of what λ-calculus is. The expression $\lambda x.f$ will denote the function that to an input a gives the output $f[a/x]$, i.e. the interpretation of the expression f when the variable x is replaced by a.

Acknowledgements

This paper is based on a talk given at the Logic Colloquium in Helsinki 2003. I am grateful to the organizing committee for inviting me.
Richard Shore suggested that I should write a paper based on the talk. Without his initiative I would never have written a paper like this.
Jens Erik Fenstad, John Longley and Many Lerman gave valuable comments on a first draft for the paper.

2 The Old Days

It is normally impossible to tell when the history of a scientific dicipline starts, and the computability theory of functionals of finite types is no exeption. For the purpose of this paper, the subject started in 1959 with the publication of three important papers, two by Kleene and one by Kreisel. We choose to begin our survey with 1959 even if e.g. λ-calculus is based on the idea of computing with functions and was developed before 1959. Moreover, the theory of hyperarithmetical sets and Π^1_1-sets is reccognized as a theory of computing,

[2]The search for even the near history of the subjects we discuss in this paper is an overwhelming task, a view also expressed in Scott [34] with reference to the time passed since the writing of Scott [33].

where quantification over $\mathbb{N}$ was seen as effective, and certainly belongs to our subject in an historical perspective.

In [11] Kleene considered the full hierarchy of functionals of finite, pure types. $Type(0)$ will be the natural numbers and inductively, $Type(k+1)$ will be the set of all total functions

$$F : Type(k) \to \mathbb{N}.$$

Using nine schemes, $S1 - S9$, Kleene gave a grand inductive definition of the relation

$$\{e\}(F_1, \ldots, F_n) \simeq a$$

with the reading: *Algorithm no. e applied to functionals $F_1, \ldots, F_n$ terminates and gives a as output.*
Basic algebraic operations on $\mathbb{N}$ will be computable. We will always let a superscript denote the type, for instance to mark the type of a variable.
The scheme $S8$:

$$\{e\}(F^{k+2}, \vec{F}) \simeq F^{k+2}(\lambda x^k \{e'\}(x^k, F^{k+2}, \vec{F})),$$

where $e = \langle 8, e' \rangle$, is the scheme for oracle calls in higher types. If we can compute a function f of type $k+1$ from parameters $F^{k+2}, \vec{F}$, then uniformly in the index we can compute $F^{k+2}(f)$.

The last scheme $S9$ is the most disputable one:

$$\{\langle 9, n \rangle\}(e, \vec{F}, \vec{G}) \simeq \{e\}(\vec{F})$$

where n is the length of $\vec{F}$. $S9$ ensures the existence of a uniform enumeration of the computable partial functionals of some fixed product of types.
In Kleene's original definitions, the indices have a build-in type checking which we have omitted. Kleene's paper had a sequel in [13].

One of Kleene's motivations was to study the strength of quantification. A functional F of type $k+2$ is called *normal* if ${}^{k+2}E$ is computable in F. ${}^{k+2}E$ represents quantification over the set of objects of type k. In particular ${}^2E$ and ${}^3E$ are of interest, ${}^2E$ leading to the hyperarithmetical extension of the arithmetical sets and ${}^3E$ offering a similar extension of the analytical hierarchy. Set recursion came as a natural continuation of the investigation of normal functionals.

A set is *semicomputable* in some context if it is the domain of a partial function that is computable in the same context. Typical contexts will be Kleene-computable relative to ${}^3E$, set recursive in HC (the hereditarily countable sets) etc. Though we e.g. in set recursion give ourselves the computational power to quantify over computable subsets of some given sets, it does not follow that we may search over semicomputable subsets of the same sets. As Moschovakis pointed out, it will be common that semicomputability does not correspond to Σ_1. Thus the investigation of computability in normal functionals and set recursion led to a finer analysis of transfinite definability than Kripke-Platek

set theory, and to a better understanding of search and selection mechanisms in general.
Among the important contributors to this school of computability we mention Gandy, Grilliot, Moschovakis, Hinman, Kechris, Harrington, MacQueen, Sacks, Moldestad, Slaman and the author. Sacks [32] is an updated introduction to this branch of generalised computability theory.
Quantification over infinite sets is by nature discontinuous, and computations will be based on infinite amounts of information about the arguments. For the rest of this paper we will focus of computations that may be infinite, but where finitary information about the arguments will suffice to determine the outcome of a computation, and thus the computations are continuous in some sense.

Kleene [12] observed that some functionals F in his type hierarchy are *hereditarily countable* in the sense that the action of F on countable arguments of lower types can be coded by an *associate*, which will be some element of Baire Space $\mathbb{N}^{\mathbb{N}}$, i.e. a countable object. Kleene showed that if F is computable relative to countable parameters, then F itself is countable.

Simultaneously and independently Kreisel [19] proposed a hierarchy of *continuous functionals.* Kreisel's motivation was to give, in precise mathematical terms, a constructive interpretation of statements of analysis. Kreisel used equivalence classes of sets of formal neighbourhoods, with an inbuildt application operator, as his continuous functionals.

There were clear differences; Kleene's countable functionals are elements of the full type structure while Kreisel's continuous functionals are only defined on continuous arguments. Kreisel defined his hierarchy for all finite types while Kleene restricted his hierarchy to pure types. Still both authors seem to have agreed that modulo these differences the approaches were equivalent in the sense that for the pure types, Kreisel's hierarchy is the extentional collapse of Kleene's hierarchy.
Later characterisations via Kuratowski limit spaces have shown that they were right, but a direct and detailed proof of the equivalence is actually quite hard to form.
Kreisel's idea was to use methods from realizability semantics to translate a statement of analysis (second order number theory) to a statement of the form

$$\exists F \forall G \psi(F,G)$$

where F and G are functionals of suitable types. If we can choose F to be *recursive*, i.e. have a computable associate, then Φ will be constructivly true according to Kreisel.
Kreisel's paper was influential, though it did not live up to his expectations. For a large class of sentences, those void of $\exists$ and $\vee$, his concept of 'constructivly true' coincides with just 'true'. Since the truth value of some of these formulas may be changed by the use of forcing, and one does not want this for a working concept of 'constructivly true', we may say that Kreisel's program failed.

An $S1-S9$-computable functional will have a computable associate, but the converse need not neccessarily be true. Tait [36] showed that the Fan Functional has a computable associate, but is not Kleene computable. The Fan Functional is defined by[3]

$$\Phi(F) = \mu n.\forall f, g \in \{0,1\}^{\mathbb{N}}(\bar{f}(n) = \bar{g}(n) \Rightarrow F(f) = F(g)),$$

'computing' the modulus of uniform continuity of the continuous type-2 functional F over the compact $\{0,1\}^{\mathbb{N}}$.

Later, Gandy defined the Γ-functional

$$\Gamma(F) = F_0(\lambda n.\Gamma(F_{n+1}))$$

where $F_n(f) = F(n * f)$[4]. Gandy conjectured and Hyland [9] showed that Γ is not computable in the Fan Functional. Actually, there is no type three continuous functional with a computable associate that is maximal in the sense of Kleene computability. This is one of many interesting results about Kleene computability relative to countable functionals that are outside the scope of this paper.

In his thesis [29] Platek suggested another approach to higher type functionals. Instead of restricting himself to total functionals, he wanted to consider partial functionals. The partial functions from $\mathbb{N}$ to $\mathbb{N}$ are ordered by inclusion of the graphs. Platek's functions of type 2 will be partial functions with partial functions of type 1 as arguments. In addition they will be monotone; if we extend the input we will get at least as much information out. Then the functionals of type 2 can be ordered pointwise, and we restrict ourselves to the monotone functionals of type 3 etc. We call Platek's functionals the *Hereditarily Consistent* ones. Platek's notion of computability is based on typed λ-calculus extended with a fixed point operator at each type.

Platek realised that Kleene's schemes will make sense for his hereditarily consistent functionals of pure types, and he proved that for his hierarchy, Kleene's approach will be equivalent to his own.

Platek's thesis was never published, but an account of these results can be found in Moldestad [27].

Platek's functionals are not continuous in any way, and continuity is essential when functionals are relevant to computer science. However, before discussing the influence from computer science on the subject we will look at a third approach to computability in discontinuous functionals, Kleene's dialogues. In a sequence of papers, [14, 15, 16, 17, 18], Kleene wanted to repair some weaknesses of his first approach, weaknesses caused by himself insisting on only accepting total inputs.

A *dialogue* is a possibly transfinite, wellordered discussion between a functional and an input, where the functional feeds the input with arguments to see how it reacts. The reaction may be that the input starts a dialogue with the argument given before giving its output. We cannot go into details, but the idea is that $F(f)$ will be the result of a sequential, deterministic process, or rather, that the

[3] $\bar{f}(n)$ will mean $\langle f(0), \ldots, f(n-1)\rangle$ as usual.

[4] $*$ is concatenation, i.e. $[n * f](0) = n$ and $[n * f](k+1) = f(k)$.

value of the extentional F applied to the extentional f will be determined by an intentional representative for F via a dialogue with an arbitrary intentional representative for f. We will discuss the difference between extentional and intentional functions later.
The distinction between intentional and extentional functionals is not new with this approach. The new aspect is the linear, deterministic process that is going on at the intentional level. In adition to mend a weakness like the fact that with the original $S1 - S9$-concept, the partial computable functionals are not closed under composition, Kleene's ambition was that his new concept would be foundationally sound as a generalization of computability accepting functionals as inputs.

The increasing use of computers induced a need for a mathematically based foundation for the theory of programming and algorithms. For certain purposes, λ-calculus seemed to be a useful tool. In order to study the behavour of a program independently of compilers and hardware, one could interpret the program as a term in the untyped λ-calculus. Dana Scott felt uneasy about the use of λ-calculus. At the time there were no semantics for the untyped λ-calculus exept term models. In the summer of 1969 Scott wrote a paper [33] where he combined the ideas of Platek of using hereditarily consistent functionals with the ideas of Kleene and Kreisel of restricting the attention to locally finitely based functionals. He defined a typed hierarchy of partial, continuous and monotone functionals rich enough to contain fixed points, based on the combinators approach to typed λ-calculus. He observed that in real life computing, almost all data will belong to some datatype. Moreover, when we write a program or describe a procedure we aim at defining a function mapping data of one or several types to data of the same or some other types. Since it is quite common programming practise to write self-calling procedures or systems of procedures calling on each other, we need to model operations on procedures and we need to be able to extract fixed points of such operations on procedures. The picture he had would involve base types, types of partial functions between base types, types of functionals operating on functions etc. The point is that we do not need the type free calculus since every object we want to model will be in one of the data types or logical types.

Scott's *LCF - Logic for the Computable Functionals* essentially has four ingredients:

1. A term language for typed objects. This consists of typed constants and combined terms. There will be typed constants representing the two standard combinators of each relevant type. The set of aditional constants will depend on the base types (data types) modelled. If we restrict ourselves to the type of natural numbers and of boolean values there will be constants for zero, true, false, successor, definition by cases and for the totally undefined object of each type. For each type σ there will be a constant Y_σ with the intended interpretation as the least fixed point operator for functions of type $\sigma \to \sigma$. The terms are combined via application.

2. A relation symbol $\sqsubseteq_\sigma$ for each type σ. The intended interpretation of $t \sqsubseteq_\sigma s$ is that t cotains less information than s.

3. A formal logic for statements in the language described above.

4. An interpretation of each type σ as a partial ordering D_σ and correspon-ding interpretations of the terms.

LCF is inspired by Platek's thesis [29] and by the work on continuous functionals by Kleene [12] and Kreisel [19].

Robin Milner realised that LCF was suitable for automated proofs, and both the general language ML and more special theorem provers for statements about higher type computations were developed, see Milner and Strachey [26].

Scott only published his paper as a historical document [34] many years later. The reason why it was not published at once is that just after he had completed and circulated the manuscript he realized that the same method he used for the construction of the set of continuous functionals of a fixed type could be used to construct a model for the untyped λ-calculus.

Domain Theory was born.

In the view of the author, domain theory is the abstract study of sets of coherent information via finite simeqimations and completions.
A *directed complete partial ordering*, a *dcpo* for short, is a partial ordering $(X, \sqsubseteq)$ where each bounded set has a least upper bound and where each upwards directed set is bounded. An object x in a *dcpo* is *finitary* if x is bounded by an element of a directed set whenever it is bounded by the least upper bound of the set. An *algebraic domain* is a *dcpo* where each object is the least upper bound of its finitary subobjects. An algebraic domain is *separable* if the set of finitary objects is countable. A separable domain is *effective* if the structure of finitary objects has an effective enumeration. In an effective domain, an object α is *computable* if the set of indices for the finitary subobjects of α is *c.e.*[5].
The alternative approach favoured by the author is to start with the finitary elements. For concrete applications, like Scott's D_σ, which actually are effective, separable algebraic domains, the finitary elements can be described in a direct way. Other examples may be the set of finite partial functions from $\mathbb{N}$ to $\mathbb{N}$ ordered in the natural way, the set of finite subsets of $\mathbb{N}$ ordered in the natural way and the set of closed rational intervals $[p, q]$ on the real line with the reversed ordering. The only general requirement will be that the finitary objects form a partial ordering where each finite bounded set will have a least upper bound. Given an ordering like this, we may form the corresponding domain as the ideal completion, identifying the original objects with the prime ideals[6]. All algebraic domains can be constructed this way up to isomorphism.

[5]*c.e.* stands for *computably enumerable*, which is *r.e.* in the old fashioned terminology.
[6]An ideal will be a set α such that

- $x \in \alpha \wedge y \sqsubseteq x \to y \in \alpha$.
- $x \in \alpha \wedge y \in \alpha \to \{x, y\}$ is bounded in α.

Let $(X, \sqsubseteq)$ be an algebraic domain. In the *Scott topology* on $(X, \sqsubseteq)$, the sets $\{x \mid x_0 \sqsubseteq x\}$ will form a basis when x_0 range over the finitary sets. A *morphism* will be a function from one domain to another that is continuous with respect to the Scott topologies. In this way the separable algebraic domains form a cartesian closed category with maximal elements, the so called universal domains.
The Scott topology is never Hausdorff (exept in one uninteresting case), but it is T_0. The use of topology in this context is not advanced, but the language of topology is handy. For some applications we will be interested in quotient topologies inherited from subspaces of domains, and then even problems related to the Scott topology may be nontrivial.

In footnote 1) we defined a computer scientist to be someone working at a computer science department. Experience shows that in our context there is another distinction between a computer scientist and someone working in generalised computability theory. While we in generalised computability theory ask for the best concept of computability related to a given mathematical structure, the computer scientist asks for the best mathematical structure related to a given concept of computation.
Plotkin [30] viewed Scott's LCF as a programming language PCF, a typed λ-calculus with constants for the fixed point operators. The LCF-terms will be viewed as programs, but the logic is replaced by an operational semantics on terms, in the form of conversion rules. Standard α- and β-conversion for typed λ-calculus is an integrated part. There are also conversion rules for the constants, e.g.

$$Y_\sigma t \to t(Y_\sigma t)$$

where t is a term of type $\sigma \to \sigma$ and $Y_\sigma t$ will be a term of type σ.

Executing a program will be the same as searching for a normal form of the term, using the conversion rules. As long as we have the Church-Rosser property, this can be viewed as deterministic.

The typed Scott-continuous functionals may still be used as a model for PCF interpreting the terms in the way Scott did. One of Plotkin's key results is that if we have a term that will be interpreted as a number or as a boolean value in the Scott semantics, then it reduces to the corresponding normal form by the conversion rules of PCF.

The Scott continuous functionals will only be one of many possible structures offering an interpretation of PCF-programs (or terms). A model for a typed λ-calculus is called *complete* if every computable element of the model is the interpretation of a λ-term. Thus the model will, in some sense, not contain any junk. Two closed terms t_1 and t_2 of type σ are *observationally equivalent* if for every term $s[x_\sigma]$ of type 0 (the type of the natural numbers) and every $k \in \mathbb{N}$, $s[t_1]$ reduces to (the numeral for) k if and only if $s[t_2]$ reduces to k. This means that the calculus cannot distinguish between the terms. A model is *fully abstract* if two terms that are observationally equivalent are interpreted in the same way. This means that the model is faithful to the calculus in the sense that equivalence between terms with respect to the operational semantics will coincide with equality with respect to the denotational semantics. This is another way of expressing that the model contains no junk.

Plotkin [30] showed that the partial continuous functionals do not form a complete or fully abstract model for PCF. Consider the finite function $f : \{0,1,\bot\}^3 \to \{0,1,\bot\}$ defined by

$$f(0,x,y) = x\ ,\ f(1,x,y) = y\ ,\ f(\bot,x,x) = x.$$

This simple gadget is not PCF-definable. PCF-definable functions are in some sense sequential, i.e. there are deterministic evaluation procedures for each term of type 0. However, f defined above cannot be evaluated in a deterministic way. A deterministic procedure for $f(z,x,y)$ will request defined values for a fixed subset of $\{z,x,y\}$, and for the f given there will be no such subset that can be used by an algorithm.

Adding a constant for parallell OR (much like our f) and a constant for continuous quantification over $\mathbb{N}$, Plotkin obtained a calculus for which the continuous functionals are complete and fully abstract. Plotkin [30] also showed that the partial continuous functionals are *adequate* for PCF, i.e. if a closed term of type 0 is interpreted as a natural number k, then it evaluates to k by the operational semantics for PCF.

Milner [25] used a term-model construction and produced an algebraic domain that serves as a fully abstract and complete model for PCF. Milner further proved that two such models will be isomorphic. For many years the problem of finding a conceptually well based characterisation of Milner's model was considered to be one of the most important problems related to the semantics of functional computations. The problem is known as the *full abstraction problem.*

3 The 90'ies

Kleene's schemes $S1 - S9$ was designed to define computability of functionals in the full type hierarchy, but as we have seen, they may be interpreted over Platek's hereditarily consistent functionals as well. Of course, we may interpret $S1 - S9$ over the Scott-continuous functionals too. Doing this, Ulrich Berger observed, using Platek's argument, that $S1 - S9$ will be equivalent to PCF in this setting.

By recursion on the type we may isolate the hereditarily total functionals in the Scott hierarchy. This will not be an extentional hierarchy, there may be two different functionals that are equal when restricted to total arguments. But by recursion on the type once more, we may identify extentionally equivalent total functionals. The resulting hierarchy, as essentially shown by Ershov [3, 4, 5], is isomorphic to the Kleene-Kreisel continuous functionals. Thus every Kleene-Kreisel continuous functional is represented by several hereditarily total Scott-continuous functionals. This gives two ways of computing a Kleene-Kreisel functional using $S1 - S9$, either directly like Kleene did or indirectly by computing a representative in the Scott-hierarchy.

The easiest way to see that these approaches differ is by revisiting the Γ-functional

$$\Gamma(F) = F_0(\lambda n \Gamma(F_{n+1})).$$

In the Scott-hierarchy Γ will be the solution to a functional equation which has a PCF-definable solution using the least fixed point operator. This solution will be a $S1 - S9$-computable total representative for Gandy's noncomputable Γ. Working inside $S1 - S9$, Γ is defined via an index obtained from the recursion theorem for Kleene computability.. In the Scott hierarchy this index defines a representative for Γ while in the Kleene-Kreisel case it defines the everywhere undefined function.

Let us review what have been discussed so far. For the Kleene-Kreisel continuous functionals there are (at least) two natural concepts of 'computable', the *external* one of having a computable associate and the *internal* one of being $S1 - S9$ computable. Likewise, for the Scott continuous functionals we have the external concept of being a c.e. ideal of finitary objects and the internal concept of PCF-definability. In both cases, the external concept is strictly stronger than the internal one, and the internal concept for the partial functionals is strictly stronger than the corresponding one for the hereditarily total functionals.

The external approach to computability over the Scott-continuous functionals is based on effective enumerations of the finitary elements which again is based on iterated sequence numbering. If we for instance tie computability over the reals to this concept via some natural representation of the real line, we can hardly claim to have obtained any feasability results. If we, however base our study of computability on the internal concept PCF, our computable functionals are defined via something looking like a program, and further insight may be gained by analysing the algorithms involved. Thus PCF is a good model for what can be defined in a language for functional programming.

Berger [2] showed that the fan functional has a PCF-definable representative. Later it was known that Gandy (unpublished) knew this result around 1983. Simpson [35] used the method of Berger to show how Riemann integration and other continuous operators on real valued functions can be programmed in a programming language for exact computations over the reals introduced by diGianantonio [6, 7, 8]. This means that the program for $\int_0^1 f(x)dx$ will use oracle calls for $f(x)$ where we may use any program for f. f does not have to be given externally in form of e.g. a modulus for uniform continuity and exact values on rational inputs.

Finally Normann [28] showed that all Kleene-Kreisel functionals with a computable representative will have a PCF-definable representative. This result was proved in 1997. In parallell, Plotkin wrote a paper [31] on consequences of a stronger version of the theorem, seen as a conjecture. Normann's construction satisfied Plotkin's extra requirements. Thus the Kleene-Kreisel continuous functionals can be computed using only sequential means, via

Theorem(Normann [28])
Given a Scott-continuous hereditarily total functional F and an enumeration α of the finitary simeqimations to F, there is a total functional $G \sqsubseteq F$ uniformly

PCF-definable in α.

Corollary(Plotkin [31])
The extentional collapse of the total objects in Milner's model is isomorphic to the Kleene-Kreisel continuous functionals.

These results are mainly of foundational interest. If we extract a PCF-program from an external description of a functional, of course even more feasability is lost. One way to explain the results is that at least for total functionals, any program with a non-deterministic component can be replaced by a deterministic one at the cost of computational complexity and thus it generalizes a well known fact to functional programming.

Now it is about time to be more precise about the difference between intentional and extentional functions. An extentional function will only relate an argument to the output while an intentional object will contain some information about why a particular output is related to a particular input. Typical examles are Kleene dialogues. Of cours, modelling algorithms, the intentional aspect is important in the sense that we also want to find mathematical models that capture the essentials of the operational semantics of the algorithm.

Recently Longley [24] has elaborated on the proof of the theorem above, and shown that under mild and reasonable assumptions one can prove that the extentional collapse of the hereditarily total elements in an intentionally given model for PCF or a similar calculus will coincide with the Kleene-Kreisel continuous functionals. Thus our concept of computability for intentional or extentional partial objects, and our choice of partial objects used to model our chosen concept will have little influence on which hereditarily total objects there are. This again is a foundational result.[7]

The problem of finding a conceptually sound characterisation of Milner's model remains. Milner's model is by design the ideal completion of the finitary, hereditarily sequential functionals seen as extentional objects. The problem then splits into subproblems

1. Can we characterize the hereditarily finitary sequential functionals as extensional or as intensional objects?

2. Will any object in the ideal completion of the extentional finitary sequential functionals be sequential themselves in any reasonable sense?

As a problem related to 2. we may ask for a conceptually sound structure of sequential functionals.
There are successfull approaches to some of these problems. We will mention two:

1. Abramsky, Jagadeesan and Malacavia [1] use what they call *Game Semantics*. An intentional representative of a functional will be a strategy in a game. In these games there will be only winners, playing the game for

[7]We may say that Longley's result in a sense tells us that the concept of a finitely based hereditarily total functional of finite type is an absolute one.

a functional means cooperating with a player for the input in obtaining the output. Game semantics has evolved into something like a subject of its own, and is now a frequently used tool when one needs a fully abstract model of some programming language.

2. Hyland and Ong [10] modifies Kleene's dialogue semantics to a continuous setting.

The game and the dialogue semantics are equivalent, and it is shown that the extentional collapse of the finitary objects in these intentional models for PCF coincides with the finitary objects in Milner's model. Both approaches are technical and neither the game semantics nor the dialogue semantics can be explained in full here. However, using a known characterisation of the finitary objects of both type hierarchies, we may give the reader an idea of how to construct the Abramsky, Jagadeesan, Malacavia, Hyland, Ong - intentional sequentional functionals and of how to give an alternative construction of Milner's model.

Definition
We define the *type terms* by:
0 is a type term.
If σ and τ are type terms, then $(\sigma \to \tau)$ is a type term.

The intended interpretation will be $\mathbb{N}_\perp$ for 0 and some set of functions from *type*(σ) to *type*(τ) for $(\sigma \to \tau)$.
It is customary to view an object of type $(\sigma \to (\tau \to \delta))$ as a function of two variables of types σ and τ resp. We will then simplify the notation and write $\sigma, \tau \to \delta$ instead of $(\sigma \to (\tau \to \delta))$.
Expanding this convention, we see that each type can be written on the form

$$\sigma = \sigma_1, \ldots, \sigma_n \to 0$$

where $n \geq 0$.

We will now simultaneously for each σ define the *Finite Sequential Procedures* of type σ, the FSP's for short. We will let $\sigma = \sigma_1, \ldots, \sigma_n \to 0$, $\sigma_i = \delta_{i,1}, \ldots, \delta_{i,m_i} \to 0$ when $1 \leq i \leq n$ and $\delta_{i,j} = \pi_{i,j,1}, \ldots, \pi_{i,j,k_{i,j}} \to 0$ when $1 \leq i \leq n$ and $1 \leq j \leq m_i$. $\vec{F}$ will denote an unspecified ordered sequence of FSP's of types $\sigma_1, \ldots, \sigma_n$ resp.

We define the FSP's in terms of how the algorithms for them work, but technically we should define them as formal terms with interpretations in typed λ-calculus.

Definition

1. For each $a \in \mathbb{N}$, $C_a(\vec{F}) = a$ is an FSP of type σ.

2. If $1 \leq i \leq n$, Q_j is an FSP of type

$$\pi_{i,j,1}, \ldots, \pi_{i,j,k_{i,j}}, \sigma_1, \ldots, \sigma_n \to 0$$

for each $j \leq m_i$, $B \subset \mathbb{N}$ is finite and P_b is an FSP of type σ for each $b \in B$, then we define an FSP P of type σ by the following procedure for $P(\vec{F})$:

  - Compute $b = F_i(\xi_1, \ldots, \xi_{m_i})$ where $\xi_j(\vec{x}) = Q_j(\vec{x}, \vec{F})$.
  - If $b \in B$ then $P(\vec{F}) = P_b(\vec{F})$, otherwise $P(\vec{F}) = \perp$.

3. There is one empty procedure that is everywhere undefined.

It requires a nontrivial proof to show that the intermediate ξ_j's will be finite sequential procedures. It is easy to see by induction on the combined complexity of P and $F_1, \ldots, F_n$ that we may evaluate $P(F_1, \ldots, F_n)$ in a sequential manner and in a finite number of steps.

There are two natural ways to define a partial ordering on the set of FSP's of type σ, the intentional ordering and the extentional ordering. The set of intentional extensions is defined by recursion on the formation of the procedures. There is no proper extension of a constant procedure, and a constant procedure is only extending the totally undefined procedure.
If i, $\{Q_j\}_{j \leq m_i}$, B and $\{P_b\}_{b \in B}$ are as above, we may extend the constructed P by

- keeping i
- extending each Q_j to some Q'_j
- extending B to a superset B'
- extending each P_b to some P'_b for $b \in B$
- adding some P'_b for each $b \in B' \setminus B$.

If we consider the completion of this partial ordering for each type, we get a typed hierarchy of sequential functionals, and these will correspond to the ones obtained by game semantics or dialogue semantics. If P and $\vec{F}$ are infinite procedures constructed this way, we see that $P(\vec{F})$ is determined by a sequential process that either terminates with an output a, terminates with an output $\perp$ (if we ask for some P_b that does not exist) or does not terminate at all. In this case we also let the output be $\perp$.

We define the extentional preordering simply by $P_1 \sqsubseteq P_2$ if for all $\vec{F}$ and $a \in \mathbb{N}$, $P_1(\vec{F}) = a \Rightarrow P_2(\vec{F}) = a$.
We may prove that each FSP will be monotone with respect to this preordering, and the finitary elements in Milner's model will correspond to the ordered set of classes of pre-equal elements. We then get Milner's model as the ideal completion.

We have seen how Milner's model can be obtainded as the ideal completion of the extentional collapse of the FSP's. The problem is that taking the extentional collapse is not an effective process. We actually have undecidability results:

In *finitary* PCF we restrict the natural numbers to a finite subset $\{0, \ldots, k\}$.

Theorem(Loader [20])
Unary PCF *is decidable.*

Theorem(Loader [21])
Finitary PCF *is undecidable.*

Loader shows how termination in a semi-Thue system can be reduced to termination of a term in tertiary PCF.

Corollary
The ordered set of finitary elements in Milner's model is not decidable.

Corollary
The set of finitary Kleene-computable elements of the Scott-hierarchy is undecidable.

The total complexities of these sets are unknown. The second set is known to be Σ^0_3. It is a reasonable guess that Milner's model itself is not even arithmetical. In any case we may conclude that Milner's model is not accessible by the tools of computability theory, and that the intentional models are the ones to use when we need some denotational semantics for hereditarily programmable functionals.

The remaining question now is if the extentional ordering of the sequential functionals as we have constructed them is a directed complete ordering. Since Milner's model is not decidable, this may of course be of less interest to computer science. From a mathematical point of view, however, it still looks like a challenging problem.

4 Where to go now?

For the most, our story was based on events that took place in the 20^{th} century. The interest in computability in higher types did not stop at New Years Eve of 1999 though. In particular, an interesting problem, suggested by Longley, is if the concept of an intentional, but sequential functional of higher types over the natural numbers is an absolute one, see footnote 7. However, much of the focus have moved to typed structures over other base types such as the real line. This part of our subject has not matured yet, and there are many competing approaches to computability over topological base types see e.g. Tucker and Zucker [37]. Even if we restrict our attention to PCF-like languages and the use of domain representations, the picture of what is the best model is unclear. Some of the natural questions about equivalence of approaches even turn into hard topological problems. nvertheless we expect that the subject will evolve further into this still young century.

References

[1] S. Abramsky, R. Jagadeesan and P. Malacaria, *Full abstraction for PCF* (Extended abstract), in M. Hagiya and J.C.Mitchell (eds.) Theoretical aspects of Computer Software, Springer-Verlag (1994), 1-15.

[2] U. Berger, *Totale Objekte und Mengen in der Bereichtheorie* (in German), Thesis, München 1990.

[3] Yu. L. Ershov, *Computable functionals of finite type*, Algebra and Logic 11 (1972), 203-277

[4] Yu. L. Ershov, *The Theory of Numerations*, Vol 2 (in Russian), Novosibirsk (1973)

[5] Yu. L. Ershov, *Maximal and everywhere defined functionals*, Algebra and Logic 13 (1974), 210-225

[6] P. Di Gianantonio, *A Functional Approach to Computability on Real Numbers*, Thesis, Università di Pisa - Genova - Udine, (1993).

[7] P. Di Gianantonio, *Real Number Computability and Domain Theory*, Information and Computation, Vol. 127 (1996) pp. 11 - 25.

[8] P. Di Gianantonio, *An abstract data type for real numbers*, Theoretical Computer Science Vol. 221 (1999) pp. 295 - 326.

[9] J.M.E. Hyland, *Recursion on the countable functionals*, Dissertation, Oxford 1975

[10] J. M. E. Hyland and C.-H. L. Ong. *On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model*, Information and Computation 163 (2000), 285-408.

[11] S.C. Kleene, *Recursive functionals and quantifiers of finite types I*, T.A.M.S. 91 (1959), 1-52.

[12] S.C. Kleene, *Countable functionals*, in A. Heyting (ed.) Constructivity in Mathematics, North-Holland (1959), 81-100.

[13] S.C. Kleene, *Recursive functionals and quantifiers of finite types II*, T.A.M.S. 108 (1963), 106-142.

[14] S.C. Kleene, *Recursive functionals and quantifiers of finite types revisited I*, in J.E. Fenstad, R.O. Gandy and G.E. Sacks (eds.) Generalized Recursion Theory II, North-Holland (1978), 185-222.

[15] S.C. Kleene, *Recursive functionals and quantifiers of finite types revisited II*, in J. Barwise, H.J. Keisler and K. Kunen (eds.) The Kleene Symposium, North-Holland (1980), 1-29.

[16] S.C. Kleene, *Recursive functionals and quantifiers of finite types revisited III*, in G. Metakides (ed.) Patras Logic Symposion, North-Holland (1982), 1-40.

[17] S. C. Kleene, *Unimonotone functions of finite types (Recursive functionals and quantifiers of finite types revisited IV)*, in A. Nerode and R.A. Shore (eds.) *Recursion theory*, AMS Proceedings of symposia in pure mathematics 42 (1985), 119-138.

[18] S.C. Kleene, *Recursive functionals and quantifiers of finite types revisited V*, T.A.M.S. 325. (1991), 593-630.

[19] G. Kreisel, *Interpretation of analysis by means of functionals of finite type*, in A. Heyting (ed.) Constructivity in Mathematics, North-Holland (1959), 101-128

[20] R. Loader, *Unary PCF is Decidable*, Theor. Comput. Sci. 206(1-2), (1998), 317-329.

[21] R. Loader,*Finitary PCF is not decidable*, Theor. Comput. Sci. 266(1-2), (2001), 341-364.

[22] J.R. Longley, *The sequentially realizable functionals*, Annals of Pure and Applied Logic, vol. 117 no. 1, 1 - 93.

[23] J.R. Longley, *Notions of computability at higher types I*, To appear in Proceedings of Logic Colloquium 2000 (Paris), A.K. Peters.

[24] J.R. Longley, *On the ubiquity of certain total type structures*, To appear in Proceedings of Workshop in Domains VI (Birmingham), Electronic Notes in Theoretical Computer Science vol. 74, Elsevier, 2003.

[25] R. Milner, *Fully abstract models for typed λ-calculi*, Theoretical Comp. Sci. 4, (1977) 1 - 22.

[26] R. Milner and C. Strachey, A theory of Programming Language Semantics, Chapman and Hall, London 1976.

[27] J. Moldestad, *Computations in Higher Types*, Lecture Notes in Mathematics 574, Springer-Verlag, 1977.

[28] D. Normann, *Computability over the partial continuous functionals*, Journal of Symbolic Logic 65 (2000), 1133-1142.

[29] R. A. Platek, *Foundations of Recursion Theory*, Thesis, Stanford University, 1966.

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

[31] G. Plotkin, *Full abstraction, totality and PCF*, Math. Struct. in Comp. Science (1999), vol. 11, pp. 1-20.

[32] G. E. Sacks, *Higher Recursion Theory*, Springer verlag, 1990.

[33] D. Scott, *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Unpublished notes, Oxford (1969)

[34] D. Scott, *A type-theoretical alternative to ISWIM, CUCH, OWHY*, Theoretical Computer Science 121 (1993), 411 - 440.

[35] A. Simpson,*Lazy Functional Algorithms for Exact Real Functionals*, Mathematical Foundations of Computer Science 1998, Springer LNCS 1450, 456-464, 1998.

[36] W.W. Tait, *Continuity properties of partial recursive functionals of finite type*, unpublished notes (1958).

[37] J. V. Tucker and J. I. Zucker, *Abstract versus concrete models of computation on partial metric algebras*, ACM Transactions on Computational Logic, accepted.